Jyrki Vesterinen

Verkkopalvelun jatkokehittäminen

Ruby on Rails -ohjelmistokehyksellä

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

28.4.2013

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Jyrki Vesterinen Verkkopalvelun jatkokehittäminen Ruby on Rails -ohjelmisto-kehyksellä 57 sivua 28.4.2013
Tutkinto	insinööri (AMK)
Koulutusohjelma	tietotekniikan koulutusohjelma
Suuntautumisvaihtoehto	ohjelmistotekniikka
Ohjaaja(t)	partner Samuel Schubak yliopettaja Erja Nikunen

Tässä opinnäytetyössä tutkin, kuinka helppoa on jatkokehittää olemassa olevaa Ruby on Rails -ohjelmistokehyksellä kehitettyä verkkopalvelua. Palvelu, jota jatkokehitin opinnäytetyötä tehdessäni on nimeltään Ohjelmasuomi. Tutkin paitsi Rails-ohjelmistokehyksen, myös Ruby-ohjelmointikielen ja Railsin kanssa yleisesti käytettyjen kirjastojen, kielten ja teknologioiden vaikutusta jatkokehitettävyyteen.

Työssä kerron Ruby-ohjelmointikielestä ja annan ohjeita sen käyttöön. Sen jälkeen kuvaan Rails-ohjelmistokehystä ja annan ohjeita sen käytöstä. Sitten kerron RSpec-testityökalusta. Sen jälkeen kuvaan lyhyesti Sass-tyylitiedostokieltä ja jQuery-kirjastoa, joita käytetään Ohjelmasuomen näkymissä. Lopuksi arvioin näiden kirjastojen, kielten ja teknologioiden vaikutusta jatkokehitettävyyteen.

Päädyin tulokseen, että Ruby on Rails on jatkokehitettävyydeltään epätasainen kokonaisuus. Ainakin jos Ajaxia käytetään runsaasti, on vaikeaa seurata, mitä CoffeeScript-skriptit oikeastaan tekevät. Lisäksi ainakin Ohjelmasuomen koodissa hyödynnetään Railsin implisiittisiä lisäyksiä, mikä vaikeuttaa koodin seuraamista pahasti. Palvelinpuolen koodissa Rails on huomattavasti parempi, mutta en voi suositella Railsia pitkiin projekteihin.

Avainsanat	ruby, rails, ruby on rails, jatkokehittäminen

Author(s) Title Number of Pages Date	Jyrki Vesterinen Further Developing a Ruby on Rails -Based Website 57 pages 28 April 2013
Degree	Bachelor of Engineering
Degree Programme	Degree programme of information technology
Specialisation option	Software engineering
Instructor(s)	Partner Samuel Schubak Principal Lecturer Erja Nikunen

The aim of the study was to investigate how easy it is to further develop an existing website created with the Ruby on Rails framework. The site developed for the study is called Ohjelmasuomi. The study also investigates how the Ruby programming language and some libraries, languages and technologies commonly used with Rails affect the effectiveness of further developing a website.

In the study, the Ruby programming language is introduced and some usage instructions are presented. After that, the Rails framework is described and again, usage instructions are given. Then the RSpec testing tool is explained. Next the Sass stylesheet language and the jQuery library, used in the views of Ohjelmasuomi, are briefly described. At the end, it is evaluated how these libraries, languages and technologies affect maintainability.

The study would suggest that Ruby on Rails is not uniform regarding the effectiveness of further developing a website. At least if Ajax is heavily used, it is difficult to follow what the CoffeeScript scripts are actually doing. In addition, Ohjelmasuomi uses Rails's implicit additions which make the code a lot harder to follow. Rails is significantly better at server-side code, but it can't be recommended for long-term projects.

Keywords	Ruby, rails, ruby on rails, further developing

Sisällys

Lyhenteet

Ajax — Asynchronous JavaScript And XML. Joukko websovelluskehitystekniikoita, joilla websovelluksista voi tehdä vuorovaikutteisempia.

ankkatyypitys — *Dynaamisen tyypityksen* tyyli, jossa sallittu semantiikka riippuu olion operaatioista ja määreistä eikä perityistä luokista tai toteutetuista rajapinnoista.

asettelu — *Näkymä*, jonka sisään kaikki muut *näkymät* piirretään.

ASP — Active Server Pages. Microsoftin kehittämä dynaamisten *WWW*-sivujen luomiseen tarkoitettu palvelinpuolen ohjelmointimenetelmä.

auttaja — *Moduuli*, joka tarjoaa lyhyitä metodeja *näkymän* käyttöön.

CoffeeScript — Skriptikieli, joka käännetään *JavaScriptiksi*. Lisää *JavaScriptiin* syntaksista karkkia (syntactic sugar) ja uusia ominaisuuksia.

CRUD — Create, Read, Update and Delete. Pysyvän tallennuksen neljä perusoperaatiota.

CSRF — Cross-site Request Forgery. Hyökkäys, jossa hyökkääjä lähettää komentoja luotetulta käyttäjältä ilman käyttäjän lupaa.

CSS — Cascading Style Sheets. Erityisesti *WWW*-sivuille suunniteltu tyylitiedostokieli.

CSS-valitsinmoottori — Ohjelmisto, joka pystyy löytämään dokumentista *valitsinta* vastaavat elementit.

dokumenttipuu

Puu, joka esittää dokumenttia.

DOM Document Object Model. Rajapinta, jolla voi muokata *HTML*- ja *XML*-dokumentteja. Esittää dokumentin *dokumenttipuuna.*

double Valeolio, jota käytetään todellisen olion tilalla *yksikkötesteissä.*

Drupal *PHP*-kielellä kirjoitettu avoimen lähdekoodin sisällönhallintajärjestelmä.

DRY Don't Repeat Yourself. Ohjelmistokehityksen periaate, joka pyrkii vähentämään informaation toistamista.

dynaaminen tyypitys

Tyyppijärjestelmä, jossa olioiden tyypit voivat muuttua suoritusaikana.

ERB Embedded Ruby. Merkintäkieli, jolla *Ruby*-koodia voi upottaa tekstitiedostoon.

Haml HTML Abstraction Markup Language. Kevyt merkintäkieli, joka käännetään *HTML*:ksi.

hierarkkinen Jakaa alkiot tasoihin (level).

HTML Hypertext Markup Language. Merkintäkieli, jolla useimmat *WWW*-sivut on kirjoitettu.

HTML-entiteetti

Token, jonka *HTML*-tulkki korvaa Unicode-merkillä.

HTTP Hypertext Transfer Protocol. Protokolla, jota *WWW*-selaimet ja palvelimet käyttävät tiedonsiirtoon.

instanssimuuttuja

Oliokohtainen muuttuja.

Java — Yleiskäyttöinen, rinnakkainen oliopohjainen ohjelmointikieli, jonka suunnittelussa on tavoiteltu, että toteutuksilla on mahdollisimman vähän riippuvuuksia.

JavaScript — Skriptikieli, jota käytetään *WWW*-sivujen käyttöliittymän parantamiseen. Ei liity *Javaan.*

jQuery — Suosittu *JavaScript*-kirjasto.

JRuby — *JVM*:ään perustuva *Ruby*-tulkki.

JSON — JavaScript Object Notation. Yksinkertainen tiedonsiirtomuoto, jota on helppo käyttää *JavaScript*-skripteissä.

JVM — Java Virtual Machine. Virtuaalikone, joka voi suorittaa *Java*-tavukoodia.

kasa — Tietorakenne, josta prosessi varaa muistia.

keräytymäfunktio

Funktio, jossa useiden rivien arvoista muodostetaan yksi arvo.

koodiavaruus

Arvojoukko, jota käytetään kirjoitusmerkkien koodaamiseen.

koodipiste — Yksi numeroarvoista, joista *koodiavaruus* koostuu.

kutsupino — *Pino*, jossa säilytetään tietoa säikeen aktiivisista aliohjelmista.

literaali — Merkintätapa, jolla vakion voi esittää lähdekoodissa.

malli — *Model View Controller* -suunnittelumallissa ohjelmiston osa, joka tallentaa ja käsittelee tietoa.

mallinemoottori

Webmallinejärjestelmän osa, joka käsittelee *webmallineita* ja sisältötietoja tuottaakseen webdokumentteja.

Markdown Kevyt merkintäkieli, joka painottaa voimakkaasti ihmisluettavuutta (human readability).

MIT Massachusetts Institute of Technology. Amerikkalainen yliopisto, joka on luonut *MIT-lisenssin.*

MIT-lisenssi Vapaa ohjelmistolisenssi, jonka *MIT* on luonut.

mock *Double*, joka tarkistaa, että sen operaatioita kutsutaan.

mockaus *Mock*ien käyttö *yksikkötesteissä.*

Model View Controller

Suunnittelumalli, jossa ohjelmisto jaetaan *malliin* (model), *näkymään* (view) ja *ohjaimeen* (controller).

modulaarisuus

Suunnittelumalli, jossa ohjelma jaetaan toisistaan riippumattomiin ja korvattavissa oleviin *moduuleihin.*

moduuli Metodi- ja vakiokokoelma tai *modulaarisen* ohjelman osa.

muistivuoto Yleinen ohjelmointivirhe, jossa ohjelmiston osa ei vapauta varaamaansa muistia, kun sitä ei enää tarvita.

Mustache Yksinkertainen *webmallinejärjestelmä.*

näkymä *Model View Controller* -suunnittelumallissa ohjelmiston osa, joka sisältää ohjelmiston käyttöliittymän.

odotuskieli Kieli, jolla voi ilmaista, mitä ohjelmiston pitäisi tehdä *yksikkötestissä.*

ohjain *Model View Controller* -suunnittelumallissa ohjelmiston osa, joka vastaanottaa käyttäjältä tulevat käskyt ja muuttaa *mallia* ja *näkymää* käskyjen mukaisesti.

ohjauksen inversio

Ohjelmointitekniikka, jossa oliot sidotaan toisiinsa suorituksen aikana.

ohjauskoodi *Koodipiste*, joka ei vastaa yhtään kirjoitusmerkkiä.

paikallinen muuttuja

Muuttuja, joka on käytettävissä vain operaatiossa tai lohkossa, jossa se on määritetty.

pakomerkki Merkki, joka estää seuraavan merkin tulkitsemisen syntaksiksi.

partiaali *Näkymän* osa.

PHP PHP: Hypertext Preprocessor. Ohjelmointikieli, jota käytetään erityisesti palvelimissa dynaamisten *WWW*-sivujen luonnissa.

pino Tietorakenne, josta elementit poistetaan vastakkaisessa järjestyksessä niiden lisäämiseen verrattuna. Esimerkiksi viimeksi lisätty elementti poistetaan ensimmäisenä.

puu *Hierarkkinen* tietorakenne.

pseudovalitsin

Valitsin, jossa elementin valintaperuste ei ole sijainti *dokumenttipuussa.*

Python Monipuolinen, tulkattava ohjelmointikieli.

pääavain Tietokantakenttä, joka identifioi rivin.

reflektio Ohjelman kyky tutkia ja muuttaa olion rakennetta ja käyttäytymistä suoritusaikana.

reititin Ruby on Rails -ohjelmistokehyksen osa, joka tunnistaa *URL*:n ja lähettää *HTTP*-pyynnön *toiminnolle* käsiteltäväksi.

reitti Asetus, joka sitoo *URL*:n *toimintoon.*

REST Representational State Transfer. Ohjelmistoarkkitehtuuri hajautetuille järjestelmille, kuten *WWW*:lle. Hallitseva *Web service* -suunnittelumalli.

RESTful Noudattaa *REST*in periaatteita.

roskienkerääjä

Ohjelmisto, joka vapauttaa automaattisesti sellaisten muuttujien varaaman muistin, joita ohjelma ei tule enää käyttämään.

Ruby Tulkattava, dynaaminen ja dynaamisesti tyypitetty oliopohjainen ohjelmointikieli.

RubyGems Pakettienhallintajärjestelmä *Ruby*-kielellä kehitetyille kirjastoille ja ohjelmille.

Sass Syntactically Awesome Stylesheets. Kevyt tyylitiedostokieli, joka käännetään *CSS*:ksi.

Sizzle *JQuery*-projektin perustama ja käyttämä *CSS-valitsinmoottori.*

SOAP Protokolla, jonka avulla *Web service*:t voivat välittää rakenteista tietoa.

Spring *Java*-kielellä kirjoitettu avoimen lähdekoodin ohjelmistokehys ja *ohjauksen inversio* -säiliö.

SQL Structured Query Language. IBM:n kehittämä kyselykieli, jolla relaatiotietokantaan voi tehdä hakuja, muutoksia ja lisäyksiä.

SQL-injektio Hyökkäys, jossa hyökkääjä antaa tietokantapalvelimelle SQL-komentoja, joita hänen ei pitäisi pystyä antamaan.

stubbaus *Tynkien* käyttö *yksikkötesteissä.*

suodatin Metodi, joka sieppaa *toimintojen* kutsut ja tekee jotakin ennen *toiminnon* kutsumista ja/tai *toiminnon* suorittamisen jälkeen.

säännöllinen lauseke

Yksinkertainen merkkijonokieli, jolla kirjoitettu lauseke joko vastaa tai ei vastaa jotakin toista merkkijonoa.

sääntö Komponentti, joista *tyyliarkki* koostuu.

testivetoinen kehitys

Ohjelmointia tukeva tekniikka, jossa luodaan ensin uusi *yksikkötesti* ja vasta sitten muokataan ohjelmistoa niin, että se läpäisee uuden testin.

tietokantamigraatio

Skripti, joka muuttaa tietokannan rakennetta.

toiminto *Ohjaimen* osa, joka kuvaa säännön, jolla vastataan yhteen tietynlaiseen *HTTP*-pyyntöön.

toistorakenne

Kontrollirakenne, joka suorittaa sarjan toimenpiteitä yhä uudelleen niin kauan kuin jokin ehto on voimassa.

token Merkkijono, jonka voi kategorisoida tiettyjen sääntöjen mukaan.

transaktio Looginen operaatiokokonaisuus tietokannan käsittelyssä.

tynkä *Double*, jonka operaatiot palauttavat etukäteen päätettyjä arvoja.

tyyliarkki *CSS*-tiedosto.

unkarilainen notaatio

Nimeämiskäytäntö, jossa muuttujan nimi ilmaisee sen tyypin tai käyttötarkoituksen.

URL Uniform Resource Locator. Merkkijono, joka viittaa tiettyyn Internetissä olevaan resurssiin.

valitsin *Säännön* osa, joka määrittää, mitä elementtejä *sääntö* koskee.

Web service *WWW*-pohjainen ohjelmointirajapinta.

webmalline *Webmallinejärjestelmän* osa, jota käytetään webdokumenttien massatuotantoon ja sisällön erottamiseen esitysmuodosta.

webmallinejärjestelmä

Ohjelmisto, joka tuottaa webdokumentteja käsittelemällä *webmallineita mallinemoottorilla.*

WWW World Wide Web. Internetissä toimiva hajautettu hypertekstijärjestelmä.

XHTML Extensible Hypertext Markup Language. *HTML*:stä kehitetty merkintäkieli, joka täyttää *XML*:n muotovaatimukset.

XML Extensible Markup Language. Ihmisen ja koneen luettavissa oleva dokumenttien merkintäkieli.

yksikkötesti Koodinpätkä, joka testaa automaattisesti ohjelmiston osan toimintaa.

1 Johdanto

Ruby on Railsilla (1) on hyvä maine ohjelmistokehyksenä, jolla saa nopeasti verkkopalvelun aikaan (2). Toisaalta se skaalautuu jopa todella suuriin sivustoihin: tunnetuin Railsiin perustuva WWW-sivusto on Twitter (3).

Aiemmin kuitenkaan ei ole pohdittu, kuinka helppoa on jatkokehittää olemassa olevaa Ruby on Railsilla kehitettyä verkkopalvelua. Esimerkki potentiaalisesta ongelmasta on Ruby-ohjelmointikielen dynaaminen tyypitys, jonka vuoksi lähdekoodissa ei lue, minkä tyyppisiä käytettävät muuttujat ovat. Vaikeuttaako se jatkokehittämistä?

On myös kirjastoja, kieliä ja teknologioita, joita käytetään yleisesti Railsin kanssa. Herää kysymys, miten ne vaikuttavat jatkokehitettävyyteen.

Tämä insinöörityö pyrkii vastaamaan yllä oleviin kysymyksiin. Insinöörityön tilaaja on viestintätoimisto CRE8 Oy (4) ja palvelu, jota jatkokehitin, on Ohjelmasuomi (5). Työn arvo tilaajalle on yksinkertaisesti jatkokehitys.

Lukijalle tämä insinöörityö antaa tietoa, kannattaako uusi verkkopalvelu laatia Ruby on Rails -ohjelmistokehyksellä, jos tiedetään, että palvelua tulevat myöhemmin kehittämään eri ihmiset. Korostan kuitenkin, että päätöstä tehtäessä on syytä ottaa huomioon muutkin tekijät, kuten alkuvaiheen kehitettävyys, tietoturva ja suorituskyky.

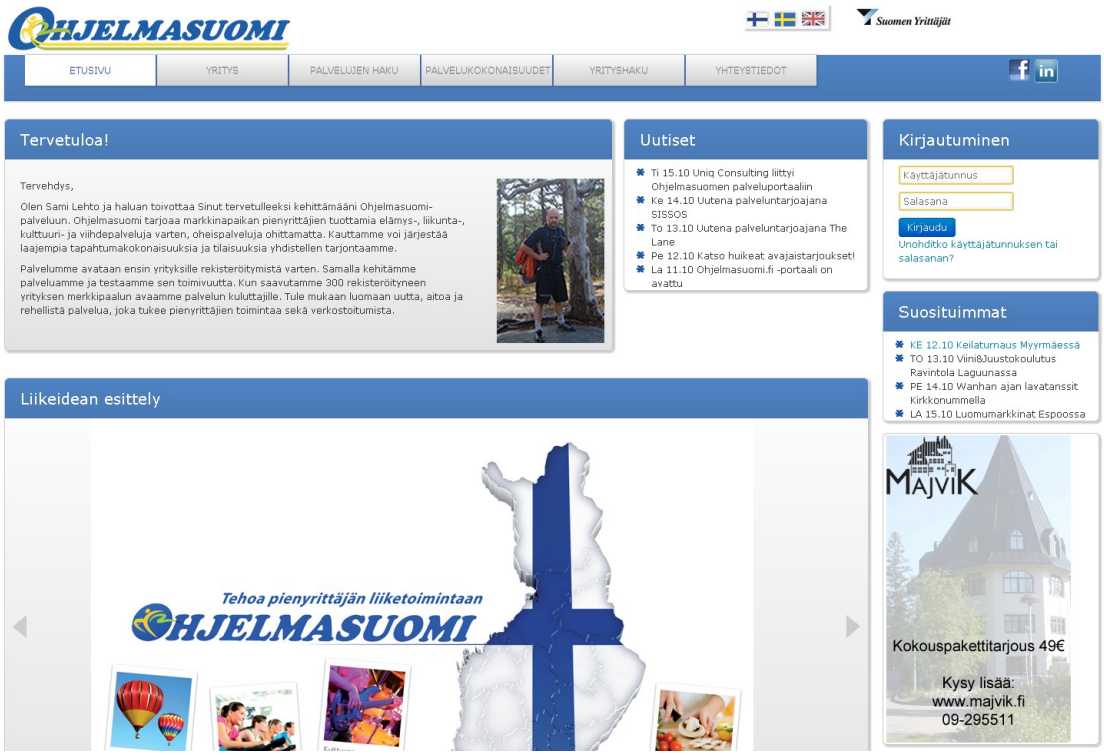
2 Verkkopalvelu

2.1 Viestintätoimisto CRE8 Oy

Viestintätoimisto CRE8 Oy on vuonna 2007 perustettu yhtiö, joka tarjoaa tiedotus- ja yhteydenpitoratkaisuja erityisesti pienille ja keskisuurille yrityksille sekä yhteisöille. CRE8 on erikoistunut sähköiseen maailmaan, kotisivujen rakentamiseen ja kehittämiseen, tietotekniikkaan sekä markkinoinnin ja yhteydenpidon integroimiseen muihin tietojärjestelmiin. (4.)

2.2 Ohjelmasuomi

Ohjelmasuomi on ohjelmapalvelujen markkinapaikka, jonka tavoite on saattaa kuluttajat ja palveluntarjoajat yhteen. Ohjelmasuomi pyrkii ensisijaisesti ratkaisemaan monia palveluntarjoajia vaivaavan ongelman; kuluttajien löytämiseksi on pystytettävä WWW-sivusto, mikä edellyttää sivuston kirjoittamista, domainin ostamista, hostauksen hankkimista, mainostamista, jotta kuluttajat löytävät sivuston ja niin edelleen. Pienelle yritykselle se on aika paljon lisävaivaa, eikä yrityksellä välttämättä ole edes tarvittavaa osaamista. Tavoitteena on, että tulevaisuudessa palveluntarjoaja voi yksinkertaisesti rekisteröityä Ohjelmasuomeen kuukausimaksua vastaan ja saada asiakkaita sitä kautta.



Kuva 1. Ohjelmasuomi. Kuvassa näkyvä data, kuten uutisten otsikot, on testidataa.

Ohjelmasuomen on tarkoitus helpottaa palveluntarjoajien elämää muillakin tavoin. Tulevaisuudessa palveluntarjoajat voivat ostaa mainoksia suoraan Ohjelmasuomesta. Ne voivat sopia keskenään tekevänsä palvelukokonaisuuksia, esimerkiksi juhlien jälkeen siivousta. Palveluun tulee sisäinen ilmoitustaulu yritysten välistä kaupankäyntiä varten. Samoin yritykset voivat tehdä tarjouspyyntöjä toisilleen.

3 Jatkokehitettävyys

Keskimäärin noin 67 % kaikista ohjelmistokehityksen kustannuksista syntyy ylläpitovaiheessa (6, s. 57). Sen vuoksi ohjelmistoprojektin aikana on syytä pyrkiä minimoimaan ylläpitovaiheen kustannukset.

Standardi ISO/IEC 25010, Systems and Software Quality Requirements and Evaluation antaa ylläpidettävyydelle määritelmän, jonka mukaan ylläpidettävyyden osia ovat modulaarisuus, uudelleenkäytettävyys, analysoitavuus, muokattavuus ja testattavuus. (7.)

Käsite	Määritelmä
Ylläpidettävyys	Kuinka tehokkaasti ja tuloksellisesti kehittäjät voivat muokata tuotetta tai järjestelmää.
Modulaarisuus	Missä määrin järjestelmä tai tietokoneohjelma koostuu erillisistä komponenteista, niin että yhteen komponenttiin kohdistuva muutos vaikuttaa muihin komponentteihin mahdollisimman vähän.
Uudelleenkäytettävyys	Missä määrin omaisuuserää voi käyttää useassa järjestelmässä tai muiden omaisuuserien luonnissa.
Analysoitavuus	Kuinka tehokkaasti ja tuloksellisesti muutoksen vaikutusta tuotteen tai järjestelmän osiin voi arvioida, tai heikkouksia ja virheiden syitä voi etsiä, tai muokattavat osat voi löytää.
Muokattavuus	Kuinka paljon tuotetta tai järjestelmää voi muokata tehokkaasti ja tuloksellisesti tuottamatta virheitä ja heikentämättä tuotteen laatua.
Testattavuus	Kuinka tehokkaasti ja tuloksellisesti järjestelmän, tuotteen tai komponentin testauskriteerit voi perustaa ja testejä voi suorittaa sen selvittämiseksi, onko kriteerit saavutettu.

4 Ruby on Rails -ohjelmistokehys

Ohjelmasuomi on kehitetty Ruby on Rails -ohjelmistokehyksen avulla. Jotta antamani koodiesimerkit voi ymmärtää, on ensin tunnettava Ruby on Rails ainakin pintapuolisesti. Sen vuoksi käyn seuraavissa aliluvuissa läpi Ruby-ohjelmointikielen ja Rails-ohjelmistokehyksen tärkeimmät ominaisuudet.

4.1 Ruby-ohjelmointikieli

Ruby on tulkattava, dynaaminen ja dynaamisesti tyypitetty oliopohjainen ohjelmointikieli.

4.1.1 Oliopohjaisuus

Rubyssä kaikki muuttujat ja vakiot ovat olioita, ja jokainen lauseke palauttaa olion (8, s. 37). Esimerkiksi Javaa pidetään usein "aitona oliokielenä", mutta Javassa on skalaarisia muuttujatyyppejä, kuten `int` ja `boolean`, ja Java-funktioiden paluuarvon tyyppi voi olla `void`, jolloin funktiokutsu ei palauta oliota eikä edes skalaarista muuttujaa.

Sen sijaan Rubyssä voi tehdä jotakin tällaista:

```
def method
  5.times.map{ 12 + Random.rand(20) }
end
```

Kuten näkyy, 5 on olio, jonka metodeja voi kutsua täysin normaalisti. Metodissa ei lue missään "return", koska se on tarpeetonta – Rubyssä metodi palauttaa viimeisen lausekkeen paluuarvon automaattisesti. Tässä tapauksessa paluuarvo on viidestä luvusta koostuva taulukko.

Metodikutsuissa sulut ovat vapaaehtoisia, jopa silloin, kun kutsussa on parametreja (8, s. 38). Jos kirjoitan edellisessä esimerkissä sulut jokaiseen mahdolliseen paikkaan, lopputulos on:

```
def method()
  5.times().map(){ 12 + Random.rand(20) }
end
```

Rubyssä metodien ja paikallisten muuttujien nimien on pakko alkaa joko pienellä kirjaimella tai alaviivalla. Instanssimuuttujien nimet alkavat @-merkillä (esim. `@organization`). Monesta sanasta koostuvassa muuttujan nimessä on tapana erottaa sanat alaviivoilla (esim. `current_user`). (8, s. 38.)

Luokkien, moduulien ja vakioiden nimien sen sijaan täytyy alkaa isolla kirjaimella (8, s. 38). Monesta sanasta koostuvassa luokkanimessä käytetään yleensä CamelCase-tapaa (esim. `OrganizationsController`) (8, s. 38). Tällä tavoin Ruby pakottaa ohjelmoijan käyttämään unkarilaista notaatiota.

Lausekkeen perään ei tarvitse panna puolipistettä, jos jokainen lauseke kirjoitetaan omalle rivilleen. Kommentti alkaa risuaidalla (#) ja jatkuu rivin loppuun asti. Ruby, toisin kuin Python, ei välitä sisennyksestä, mutta kahdella välilyönnillä sisentäminen on de-facto standardi. (8, s. 39.)

Monissa ohjelmointikielissä merkitään aaltosuluilla, mitä lausekkeita esimerkiksi `if`- tai `while`-lause koskee. Sen sijaan Rubyssä kirjoitetaan `end` kohtaan, jossa lauseen vaikutusalue päättyy, tähän tapaan:

```
if current_user.has_role?(:admin) && params.has_key?(:id)
  @organization = Organization.where(:id => params[:id]).first
else
  @organization = current_user.organization
end
```

4.1.2 Muuttujatyypit

Vaikka kaikki muuttujat ovat olioita, Rubyssä on erikoissyntaksia tiettyjen luokkien literaalien (literal) määrittämistä varten.

Merkkijonoliteraalin voi määrittää kirjoittamalla merkkijonon yksinkertaisten tai kaksinkertaisten lainausmerkkien väliin. Jos ohjelmoija käyttää yksinkertaisia lainausmerkkejä, merkkijonoliteraalin sisältö vastaa lähes täydellisesti lainausmerkkien välissä olevia merkkejä. (8, s. 40.)

Kaksinkertaisten lainausmerkkien tapauksessa Ruby-tulkki korvaa joitakin tokeneita vastaavilla ohjauskoodeilla. Esimerkiksi `\n` korvataan rivinvaihdolla. (8, s. 40.)

Niin ikään kaksinkertaisten lainausmerkkien tapauksessa merkkijonon sisään voi kirjoittaa lausekkeen, joka suoritetaan ja jonka paluuarvo sijoitetaan merkkijonoon. Sitä selventää parhaiten esimerkki: (8, s. 40)

```
def say_goodnight(name)
  "Good night, #{name.capitalize}"
end
```

Rubyn merkkijonot ovat muuttuvia (mutable). Se helpottaa merkkijonojen käyttöä mutta toisaalta myös heikentää suorituskykyä, varsinkin silloin, kun merkkijonoa käytetään vain hetkellisesti. Merkkijonoa luotaessa sille on varattava muistia kasasta (heap), ja kun sitä ei enää käytetä, roskienkerääjän on vapautettava sen varaama muisti muistivuodon estämiseksi. (9.)

Sen vuoksi Ruby tarjoaa myös muuttumatonta (immutable) merkkijonoa vastaavan luokan, jota kutsutaan symboliksi. Symbol-luokalla on taulukko kaikista luoduista symboleista (`Symbol.all_symbols`). Aina kun symboliliteraalia käytetään, Ruby-tulkki tarkistaa, onko symboli jo taulukossa, ja jos on, palauttaa viitteen olemassa olevaan symboliin varaamatta lainkaan lisää muistia. Symboleiden varaamaa muistia ei myöskään koskaan vapauteta, koska symbolitaulukossa on viite jokaiseen symboliin. (9.)

Symboliliteraalin määrittämiseen on kaksi syntaksia: (8)

```
:admin
:"hello world"
```

Merkkijono on aika primitiivinen tietorakenne, joka voi sisältää vain merkkejä. Ruby sisältää syntaksitason tuen myös taulukoille ja Hash-nimiselle tietorakenteelle, joihin voi sijoittaa mitä tahansa. (8, s. 40.)

Taulukko ja hash ovat indeksoituja tietorakenteita. Kumpikin sisältää olioita, ja jokaiseen olioon pääsee käsiksi avaimen (key) avulla. Taulukossa avain on luku, mutta hashissa avain voi olla mikä tahansa olio. Sekä taulukko että hash kasvavat automaattisesti, kun niihin lisätään elementtejä. Taulukon ja hashin sisältämät oliot voivat olla keskenään eri tyyppisiä: esimerkiksi `[12, "virhe", 3.14]` on täysin sallittu taulukko. (8, s. 40.)

Taulukkoliteraalin voi määrittää kirjoittamalla taulukon sisällön hakasulkujen väliin pilkuilla eroteltuna, kuten JavaScriptissä. Taulukon elementteihin pääsee käsiksi kirjoittamalla indeksin hakasulkujen väliin seuraavalla tavalla: (8, s. 40)

```
a = [1, 'cat', 3.14]
item = a[0]
```

Merkkijonotaulukkojen luomiseen on myös vaihtoehtoinen syntaksi: (8, s. 41)

```
a = %w{ant bee cat dog elk}
```

Hash-literaalin voi määrittää kirjoittamalla hashin sisällön aaltosulkeiden väliin. Jokaiselle elementille on määritettävä sekä avain että arvo. (8, s. 41.)

```
inst_section = {
  :cello => 'string',
  :clarinet => 'woodwind',
  :drum => 'percussion',
  :oboe => 'woodwind',
  :trumpet => 'brass',
  :violin => 'string'
}
```

Avain on merkkien `=>` vasemmalla puolella ja arvo oikealla puolella. Samaa avainta ei voi käyttää samassa hashissa kahdesti, eli esimerkiksi avaimella `:drum` ei voi olla kahta arvoa. Avaimet ja arvot voivat olla mitä tahansa olioita. Hashin avaimina käytetään yleensä symboleja. Rails käyttää paranneltua hashia, jossa symboliavaimeen voi viitata vastaavalla merkkijonolla ja toisinpäin (ts. `params[:id]` on täsmälleen sama kuin `params["id"]`). (8, s. 41.)

Symbolien käyttäminen avaimina on niin yleistä, että Ruby 1.9:ssä tuli uusi syntaksi niitä varten. (8, s. 41.)

```
inst_section = {
  cello: 'string',
  clarinet: 'woodwind',
  drum: 'percussion',
  oboe: 'woodwind',
  trumpet: 'brass',
  violin: 'string'
}
```

Hashin elementteihin pääsee käsiksi samalla tavalla kuin taulukon elementteihin: (8, s. 42)

```
kind_of_oboe = inst_section[:oboe]
```

Jos metodin viimeinen parametri on hash-literaali, aaltosulut saa jättää pois. Railsin metodeissa viimeinen parametri on yleensä nimenomaan hash, jonne voi panna vapaaehtoisia asetuksia (8, s. 42). Seuraavat kaksi riviä tarkoittavat täsmälleen samaa:

```
redirect_to({:action => 'show', :id => product.id})
redirect_to :action => 'show', :id => product.id
```

Vielä yksi huomattava luokka, jonka literaaleille on erikoissyntaksia, on Regexp. Sen instanssit ovat säännöllisiä lausekkeita. (8, s. 42.)

Säännöllinen lauseke luodaan yleensä jommallakummalla näistä tavoista: (8, s. 42)

```
/pattern/
%r{pattern}
```

Se, täsmääkö merkkijono säännölliseen lausekkeeseen, voidaan testata =~-operaattorilla: (8, s. 42.)

```
if line =~ /Perl|Python/
  puts "There seems to be another scripting language here"
end
```

4.1.3 Logiikka

Ruby sisältää tavalliset kontrollirakenteet `if` ja `while`. Lisäksi saatavilla on `unless`, joka suorittaa koodin, jos ehto on epätosi, ja `until`, joka suorittaa koodia siihen asti, kunnes ehto on tosi. (8, s. 44.)

Ruby sisältää myös lausekemuuntimiksi (statement modifier) kutsutun ominaisuuden, joka sallii kontrollirakenteen kirjoittamisen lausekkeen perään. (8, s. 44.)

```
puts "Danger, Will Robinson" if radiation > 3000
```

Rubyn vasta-alkajat yllättää usein, että toistorakenteita käytetään Rubyssä harvoin. Niiden sijaan käytetään yleensä lohkoja ja iteraattoreita. (8, s. 44.)

Lohko on yksinkertaisesti pätkä koodia aaltosulkujen tai sanojen `do...end` välissä. Yleinen käytäntö on, että yksirivisissä lohkoissa käytetään aaltosulkuja ja monirivisissä lohkoissa `do...end`iä. (8, s. 44.)

```
{puts "Hello"}

do
  club.enroll(person)
  person.socialize
end
```

Lohkon voi antaa metodille parametrina. Niin tehtäessä lohko pannaan metodin parametrien perään: (8, s. 44)

```
verbose_greet("Dave", "loyal customer") {puts "Hi"}
```

Metodi voi suorittaa lohkon niin monta kertaa kuin haluaa käyttämällä `yield`-lauseketta. `Yield`ille voi myös antaa parametreja, jotka tullaan välittämään lohkolle. Jos lohkon halutaan ottavan parametreja vastaan, parametrit on luetteloitava lohkon alussa pystyviivojen välissä. (8, s. 44.)

Lohkoja käytetään usein iteraattoreiden kanssa. Iteraattori on metodi, joka kutsuu lohkoa kerran jokaista tietorakenteessa olevaa alkiota kohti ja antaa joka kierroksella eri alkion parametrina. (8, s. 45.)

```
animals.each {|animal| puts animal}
```

Edellinen koodiesimerkki on myös erinomainen esimerkki ankkatyypityksestä (duck typing). Koodirivi ei välitä ollenkaan, minkä tyyppinen tietorakenne `animals` on, eikä edes siitä, onko `animals` ylipäätään tietorakenne. Niin kauan kuin `animals` toteuttaa `each`-metodin, se muistuttaa tarpeeksi tietorakennetta, jotta sitä voi iteroida.

Metodi voi käyttää lohkoa nimettynä parametrina &-operaattorin avulla. (8, s. 45.)

```
def wrap &b
  print "Santa says: "
  3.times(&b)
  print "\n"
end

wrap {print "Ho! "}
```

Lohkon tai metodin sisällä lausekkeet suoritetaan peräkkäin – paitsi jos sattuu poikkeus. (8, s. 45.)

Poikkeukset ovat instansseja `Exception`-luokasta tai sen aliluokista. Poikkeuksen voi nostaa `raise`-metodilla. Poikkeuksen nostaminen saa Ruby-tulkin kerimään kutsupinoa auki (call stack unwinding), kunnes se löytää koodia, joka sanoo pystyvänsä käsittelemään poikkeuksen (8, s. 45). (Ellei sellaista löydy, Ruby-tulkki kaataa ohjelman.)

Metodin sisään tai `begin`- ja `end`-avainsanojen väliin voi kirjoittaa `rescue`-lauseen, joka nappaa tietyntyyppiset poikkeukset. (8, s. 45.)

```
begin
  content = load_blog_data(file_name)
rescue BlogDataNotFound
  STDERR.puts "File #{file_name} not found"
rescue BlogDataFormatError
  STDERR.puts "Invalid blog data in #{file_name}"
rescue Exception => e
  STDERR.puts "General error loading #{file_name}: #{e.message}"
end
```

4.1.4 Luokat ja moduulit

Seuraavana on esimerkki Ruby-luokasta. (8, s. 46.)

```
class Order < ActiveRecord::Base
  has_many :line_items

  def self.find_all_unpaid
    self.where('paid = 0')
  end

  def total
    sum = 0
    line_items.each {|li| sum += li.total}
    sum
  end
end
```

Luokkamääritys alkaa `class`-avainsanalla ja sitä seuraavalla luokan nimellä, jonka täytyy alkaa isolla kirjaimella. Edellisessä esimerkissä `Order` on `Base`-luokan, joka sijaitsee `ActiveRecord`-moduulissa, aliluokka. (8, s. 46.)

Railsissa luokkatason määrityksiä käytetään paljon. `Has_many` on `ActiveRecord`-moduulin määrittämä metodi, jota kutsutaan `Order`-luokan määrittämisen aikana. Tällaiset metodit vaikuttavat luokan toimintaan esimerkiksi lisäämällä metodeja. (8, s. 46.)

Luokan sisällä voi määrittää luokkametodeja (käytännössä sama kuin esimerkiksi Javan staattiset metodit) ja instanssimetodeja ("tavallisia" metodeja). Metodista tulee luokkametodi, jos sen nimen eteen kirjoittaa `self.` (8, s. 46). Edellisessä esimerkissä `find_all_unpaid` on luokkametodi ja `total` instanssimetodi.

Olio voi säilyttää tietoja instanssimuuttujissa. Jos muuttujan nimen eteen kirjoittaa @-merkin (esim. `@organization`), siitä tulee instanssimuuttuja. Instanssimuuttujat ovat oliokohtaisia, eli kahdella oliolla on erilliset instanssimuuttujat, vaikka ne olisivat ilmentymiä samasta luokasta. (8, s. 46.)

Instanssimuuttujiin ei pääse suoraan käsiksi olion ulkopuolelta (epäsuorasti kylläkin). Tarvittaessa Ruby voi luoda instanssimuuttujalle automaattisesti getterin ja/tai setterin: (8, s. 47)

```
class Greeter
  attr_accessor :name
  attr_reader :greeting
  attr_writer :age
```

Edellinen esimerkki luo metodit `name`, `name=`, `greeting` ja `age=`. Niitä voi käyttää seuraavaan tapaan:

```
g = Greeter.new("Barney")
old_name = g.name
g.name = "Betty"
```

Syntaksi on täsmälleen sama kuin millä esimerkiksi Javassa voi käsitellä julkisia instanssimuuttujia. Voidaan ajatella, että Rubyssä kaikki instanssimuuttujat ovat yksityisiä, mutta on mahdollista tehdä gettereitä ja settereitä, jotka saavat instanssimuuttujat näyttämään julkisilta.

Sen sijaan metodit voivat olla joko julkisia, suojattuja tai yksityisiä. Oletuksena ne ovat julkisia, mutta ne voi määrittää suojatuiksi tai yksityisiksi: (8, s. 47)

```
class MyClass
  def m1 # julkinen
  end

  protected

  def m2 # suojattu
  end

  private

  def m3 # yksityinen
  end
end
```

Toisin kuin joissakin muissa ohjelmointikielissä, Rubyssä edes toiset saman luokan instanssit eivät voi kutsua yksityisiä metodeja. (8, s. 47.)

Moduulit muistuttavat luokkia siinä, että ne sisältävät metodeja ja vakioita. Moduulista ei kuitenkaan voi luoda ilmentymää. Moduuleista on kahdenlaista hyötyä. Niitä voi käyttää eräänlaisina nimiavaruuksina: samannimiset metodit voi panna eri moduuleihin törmäysten välttämiseksi. Toiseksi moduulin avulla voi jakaa metodeja luokkien välillä. Jos luokka yhdistää (mix in) moduulin, moduulin instanssimetodeista tulee luokan instanssimetodeja. On mahdollista yhdistää moduuli moneen luokkaan, jolloin luokat jakavat moduulin metodit. Niin ikään luokka voi yhdistää monta moduulia, siinä missä vain yhden luokan voi periä. (8, s. 47−48.)

Railsin auttajat (helpers) ovat moduuleja. Rails yhdistää auttajan automaattisesti vastaavan nimiseen näkymään, jolloin näkymä voi käyttää auttajan metodeja (8, s. 48). Esimerkiksi Ohjelmasuomen koodissa on seuraava auttaja:

```
module ApplicationHelper
  def markdown(text)
    Ohjelmasuomi::Application::MARKDOWN.render(text).html_safe
  end
end
```

Sen ansiosta jokaisessa näkymässä voi käyttää `markdown`-metodia, joka muuntaa Markdown-syntaksin HTML:ksi.

4.1.5 Esimerkki

Seuraava koodi luo tietokantataulun nimeltään `products`. Taulu sisältää `title`-, `description`-, `image_url`- ja `price`-kentät sekä pari kenttää aikaleimoja varten. (8, s. 50.)

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :title
      t.text :description
      t.string :image_url
      t.decimal :price, :precision => 8, :scale => 2
      t.timestamps
    end
  end
end
```

Tarkastellaan esimerkkiä Rubyn näkökulmasta. Koodi määrittää `CreateProducts`-nimisen luokan, joka perii `Migration`-luokan `ActiveRecord`-moduulista. Luokalla on perittyjen metodien lisäksi yksi metodi nimeltään `change`. `Change` kutsuu `create_table`-metodia, joka on peritty `ActiveRecord::Migration`-luokasta, ja antaa sille parametrina tietokantataulun nimen symbolina. (8, s. 50.)

`Change` antaa `create_table`lle myös lohkon. Kun lohkoa kutsutaan, ensimmäinen sille annettava parametri saa nimen `t`. `T` toteuttaa joukon metodeja, jotka on nimetty tietokantaan tallennettavien tyyppien mukaan, kuten `string` merkkijonoja varten. Jokainen metodi yksinkertaisesti lisää kentän määrityksen luetteloon, jota `create_table` hyödyntää generoidessaan SQL-lausetta, joka luo `products`-taulun. (8, s. 50.)

Rails hyödyntää laajasti Rubyn mahdollisuuksia tehdäkseen määrityksistä mahdollisimman yksinkertaisia ja selkeitä. Edellisen tietokantamigraation voi todennäköisesti tulkita ilman Ruby- tai Rails-osaamista. Jopa pienet Rubyn ominaisuudet, kuten mahdollisuus jättää sulut ja aaltosulut pois, parantavat migraation luettavuutta ja helpottavat sen ylläpitoa. (8, s. 50.)

4.1.6 Ruby-idiomeja

Kun koodissa käytetään useita Rubyn ominaisuuksia yhdessä, voi olla epäselvää, mitä koodi oikeastaan tarkoittaa. Seuraavana on muutamia yleisiä idiomeja.

- Metodinimet kuten `empty!` ja `empty?`

  Ruby-metodien nimet voivat päättyä huuto- tai kysymysmerkkiin. Jos metodin nimi päättyy huutomerkkiin, se tarkoittaa yleensä, että metodi tekee oliolle jotakin tuhoisaa. Jos metodin nimi päättyy kysymysmerkkiin, metodi palauttaa totuusarvon. (8, s. 50.)

- `a || b`

  Lauseke evaluoi `a`:n. Ellei se ole `false` tai `nil`, lauseke palauttaa `a`:n. Muussa tapauksessa lauseke palauttaa `b`:n. Tämä on yleinen tapa palauttaa metodista jokin oletusarvo, jos ensimmäistä arvoa ei ole asetettu. (8, s. 51.)

- `a ||= b`

  Jos `a`:n arvo on `false` tai `nil`, lauseke sijoittaa `a`:n arvoksi `b`:n. Muussa tapauksessa lauseke ei tee mitään. (8, s. 51.)

- `obj = self.new`

  Joskus luokkametodin on luotava instanssi kyseisestä luokasta.

  ```
  class Person < ActiveRecord::Base
    def self.for_dave
      Person.new(:name => "Dave")
    end
  end
  ```

  Yllä oleva esimerkki toimii hyvin niin kauan kuin kukaan ei tee `Person`-luokasta aliluokkaa:

  ```
  class Employee < Person
    # …
  end

  dave = Employee.for_dave # palauttaa Person-olion
  ```

  Yllä olevassa esimerkissä `for_dave`-metodi oli kovakoodattu palauttamaan `Person`-olio, joten `Employee#for_dave` teki niin. Sen sijaan `self.new` palauttaa aina instanssin siitä luokasta, joka vastaanottaa metodikutsun (tässä tapauksessa `Employee`). (8, s. 51.)

- `lambda`

  `Lambda`-operaattori muuttaa lohkon `Proc`-olioksi. (8, s. 51.)

- `require File.dirname(__FILE__) + "/../test_helper"`

  `Require`-metodi lataa lähdekooditiedoston. Sen avulla voi sisällyttää kirjastoja ja luokkia, joita ohjelmisto tarvitsee toimiakseen. Normaalisti Ruby-tulkki etsii lähdekooditiedostoja hakemistoista, jotka on määritetty `LOAD_PATH`-ympäristömuuttujassa.

  Joskus ohjelmoijan on määritettävä tarkemmin, mikä tiedosto pitäisi sisällyttää. Se onnistuu antamalla `require`-metodille absoluuttinen tiedostopolku. Siinä kuitenkin on ongelma: ohjelmoija ei tiedä, mikä polku tulee olemaan, koska ohjelmiston käyttäjä voi panna ohjelmiston minne tahansa.

  Suhteellinen polku `require`-metodia kutsuvan ja sisällytettävän tiedoston välillä pysyy kuitenkin samana. Niin ollen absoluuttisen polun voi selvittää suoritusaikana ottamalla `require`-metodia kutsuvan tiedoston absoluuttisen polun (joka on erikoismuuttujassa nimeltään `__FILE__`), lyhentämällä polun hakemistopoluksi ja liittämällä loppuun sisällytettävän tiedoston suhteellisen polun. (8, s. 52.)

4.2 Rails-ohjelmistokehys

Ruby on Rails (tai lyhyemmin Rails) on Ohjelmasuomi-projektissa käytettävä ohjelmistokehys. Tässä luvussa käyn läpi, miten Railsin tärkeimpiä ominaisuuksia käytetään.

Ruby on Rails on avoimen lähdekoodin websovelluskehys Ruby-ohjelmointikielelle. Rails on itsessään kirjoitettu Ruby-kielellä. Ruby on Rails on ns. täyden pinon ohjelmistokehys, mikä tarkoittaa, että kehittäjä voi luoda sivuja ja sovelluksia, jotka hakevat tietoa palvelimelta, käyttävät tietokantaa ja piirtävät mallineita, asentamatta lisää ohjelmistoja. (10.)

Ruby on Rails korostaa tunnettujen suunnittelumallien, kuten Active record, Convention over Configuration, Don't Repeat Yourself ja Model-View-Controller, käyttöä. (10.)

Oletuksena Rails-malli sidotaan (map) vastaavan nimiseen tietokantatauluun. Esimerkiksi `User`-niminen malli sidotaan tietokantatauluun `users` ja tiedostoon `app/models/user.rb`. Niin halutessaan kehittäjä voi ohittaa (override) nämä käytännöt, mutta sitä ei suositella. (10.)

Ohjain (controller) on komponentti, joka vastaa HTTP-pyyntöihin. Ohjaimen tärkein tehtävä on päättää, mikä näkymä piirretään. Usein ohjaimen on kysyttävä mallilta tietoja, joita se välittää näkymälle. Ohjaimella voi olla yksi tai useampi toiminto (action). Railsissa toiminto kuvaa yleensä yhden säännön, jolla vastataan yhteen tietynlaiseen HTTP-pyyntöön. Ellei toimintoa sidota Railsin reitittimeen (router), siihen ei pääse suoraan käsiksi. Rails kehottaa kehittäjiä käyttämään RESTful-reittejä, jolloin toiminnot nimeltään create, new, edit, update, destroy, show ja index reititetään automaattisesti. (10.)

Railsin oletusasetuksilla näkymä on ERB-tiedosto. Normaalisti se käännetään HTML:ksi ajon aikana. Myös muita mallinekieliä kuten Hamlia ja Mustachea voi käyttää. (10.)

Ruby on Rails sisältää valmiiksi komponentteja, jotka helpottavat yleisiä kehitystehtäviä. Scaffoldaus (scaffolding) generoi automaattisesti osan yksinkertaisen sivuston tarvitsemista malleista ja ohjaimista. WEBrick on yksinkertainen Rubyn mukana tuleva palvelinohjelma. Rake on koontijärjestelmä (build system). Näiden työkalujen kanssa Rails on yksinkertainen kehitysympäristö. (10.)

Toimiakseen Rails tarvitsee palvelinohjelman. Rails-yhteensopivia palvelinohjelmia ovat mm. Mongrel, Lighttpd, Apache, Cherokee, Hiawatha, nginx (joko moduulina, kuten Passenger, tai CGI:n, FastCGI:n tai mod_rubyn avulla) ja monet muut. Vuonna 2008 Passenger ohitti Mongrelin suosiossa. (10.)

Alun perin Ruby on Rails käytti kevyttä SOAPia web servicejä varten. Myöhemmin se korvattiin RESTful web serviceillä. Rails käyttää tekniikkaa nimeltään Unobtrusive JavaScript (”huomiota herättämätön JavaScript”). Rails 3.1:ssä oletuksena käytettäväksi JavaScript-kirjastoksi vaihdettiin jQuery ja skriptien oletuskieleksi vaihdettiin CoffeeScript. (10.)

Rails tuottaa oletuksena HTML-sivuja ja XML-tiedostoja. Jälkimmäisiä käytetään RESTful web serviceissä. (10.)

Ruby on Rails vaatii toimiakseen Ruby 1.8.7:n, Ruby 1.9.2:n tai JRuby 1.5.2:n tai uudemman. Rails 4.0 ei tule enää tukemaan Ruby 1.8:aa. (10.)

Rails on jaettu seuraaviin paketteihin:

- ActiveRecord, olio-relaatiomuunnosjärjestelmä
- ActiveResource, tarjoaa web service:jä
- ActionPack
- ActiveSupport
- ActionMailer. (10.)

Ruby on Rails asennetaan yleensä RubyGems-pakettienhallintajärjestelmän kautta. Monissa Unix-klooneissa Railsin voi asentaa myös suoraan jakelun pakettivarastoista. (10.)

Yleensä kun Rails otetaan käyttöön (deploy), käytössä on tietokantapalvelin, kuten MySQL tai PostgreSQL, ja webpalvelin, kuten Apache Phusion Passenger -moduulin kanssa. (10.)

Saatavilla on myös Ruby on Rails -hostauspalveluja, kuten Heroku, Engine Yard ja TextDrive. (10.)

4.2.1 Tietokantakyselyt

Rails-malli sidotaan vastaavan nimiseen tietokantatauluun, ja vastaavasti jokainen mallin instanssi sidotaan tietokantariviin. Niin ollen looginen tapa lisätä tietokantaan rivi on luoda uusi instanssi halutusta mallista. (8, s. 278.)

```
an_order = Order.new
an_order.name = "Dave Thomas"
an_order.email = "dave@example.com"
an_order.address = "123 Main St"
an_order.pay_type = "check"
an_order.save
```

Mallien konstruktorit ottavat vastaan määrehashin vapaaehtoisena parametrina. Tässä hashissa avain tarkoittaa määreen nimeä ja arvo määreen arvoa. (8, s. 279.)

```
an_order = Order.new(
  :name => "Dave Thomas",
  :email => "dave@example.com",
  :address => "123 Main St",
  :pay_type => "check")
an_order.save
```

Kummassakaan esimerkissä ei annettu arvoa tietokantarivin pääavaimelle. Se nimittäin on tarpeetonta: ActiveRecord luo arvon automaattisesti ja tallentaa sen olion `id`-määreeseen oliota tallennettaessa. Ohjelmoija voi halutessaan lukea sen: (8, s. 279)

```
an_order.save
puts "The ID of this order is #{an_order.id}"
```

Konstruktori luo olion vain muistissa, joten olio on muistettava tallentaa tietokantaan. Käytettävissä on myös `create`-metodi, joka sekä luo uuden olion että tallentaa sen tietokantaan. (8, s. 279.)

```
an_order = Order.create(
  :name => "Dave Thomas",
  :email => "dave@example.com",
  :address => "123 Main St",
  :pay_type => "check")
```

Todellinen syy sille, että `new` ja `create` ottavat vastaan hashin, on, että se mahdollistaa olioiden luonnin lomakkeista saaduilla parametreilla: (8, s. 280)

```
@order = Order.new(params[:order])
```

Jos tietokannasta halutaan lukea rivejä, on määritettävä, mistä riveistä ollaan kiinnostuneita. ActiveRecordille annetaan jonkinlaisia kriteerejä, ja se palauttaa olioita, jotka sisältävät tietoja kriteerejä vastaavilta riveiltä. (8, s. 280.)

Yksinkertaisin tapa löytää tietokantarivi on pääavain. Jokainen malli toteuttaa `find`-metodin, joka ottaa parametreina pääavaimen arvoja. Jos metodille annetaan vain yksi arvo, metodi palauttaa olion, joka sisältää tietoja oikealta riviltä (tai heittää `ActiveRecord::RecordNotFound`-poikkeuksen). Jos metodille sen sijaan annetaan useita arvoja, se palauttaa olioista koostuvan taulukon (paitsi jos jotakin riviä ei löydy tietokannasta, missä tapauksessa metodi heittää `RecordNotFound`-poikkeuksen). (8, s. 280.)

```
an_order = Order.find(27)

total = Product.find(params[:product_ids]).sum(&:price)
```

Todennäköisesti yleisin tietokantahaku on niiden rivien etsintä, joissa tietyllä kentällä on haluttu arvo. Haku voi olla ”anna kaikki Daven tekemät tilaukset” tai ”hae kaikki blogiviestit, joiden aihe on ’Rails Rocks’”. (8, s. 280.)

Monilla muilla ohjelmistokehyksillä ohjelmoija joutuisi rakentamaan SQL-kyselyitä. ActiveRecord sen sijaan käyttää Rubyn dynaamisuutta tehdäkseen sen itse. (8, s. 281.)

Käytetään esimerkkinä `Order`-mallia, jonka määreitä ovat `name`, `email` ja `address`. Määreiden nimiä voi käyttää etsijämetodeissa, jotka palauttavat rivejä, joissa haluttu kenttä vastaa haluttua arvoa: (8, s. 281)

```
order = Order.find_by_name("Dave Thomas")
orders = Order.find_all_by_name("Dave Thomas")
orders = Order.find_all_by_email(params["email"])
```

ActiveRecord sallii myös SQL:n käytön. Jos halutaan luettelo Daven tilauksista, joissa maksutapa on ”po”, voidaan tehdä näin: (8, s. 282)

```
pos = Order.where("name = 'Dave' AND pay_type = 'po'")
```

Metodi palauttaa `ActiveRecord::Relation`-olion, joka sisältää ehtoja vastaavat rivit `Order`-olioihin käärittynä. (8, s. 282.)

Menetelmä toimii hyvin, jos ehdot tiedetään etukäteen. Mutta mitä jos asiakkaan nimi tulee ulkopuolelta, kuten weblomakkeesta? Yksi vaihtoehto on panna muuttujan arvo ehtomerkkijonoon: (8, s. 282)

```
# ÄLÄ TEE NÄIN!!!
pos = Order.
  where("name = '#{params[:name]}' AND pay_type = 'po'")
```

Se kuitenkaan ei ole hyvä idea, koska se mahdollistaa SQL-injektion. (8, s. 283.)

Dynaamisen SQL:n generointi kannattaa jättää ActiveRecordin huoleksi. ActiveRecord lisää SQL:ään tarvittavat pakomerkit ja estää siten SQL-injektiot. (8, s. 283.)

Jos `where`-metodille annetaan useita parametreja, Rails tulkitsee ensimmäisen parametrin mallineeksi. Siihen voi panna placeholdereita, jotka korvataan suorituksen aikana muilla parametreilla. (8, s. 283.)

Yksi tapa määrittää placeholderit on kysymysmerkkien sijoittaminen SQL:ään. Metodi korvaa ensimmäisen kysymysmerkin toisella parametrilla, toisen kysymysmerkin kolmannella parametrilla ja niin edelleen. Edellisen kyselyn voi siis kirjoittaa näin: (8, s. 283)

```
pos = Order.where(["name = ? AND pay_type = 'po'",
  params[:name]])
```

Placeholderit voi myös nimetä. Tällöin ne on sijoitettava SQL:ään muodossa `:nimi` ja vastaavat arvot on annettava hashissa, jonka avaimet vastaavat placeholdereiden nimiä: (8, s. 283)

```
pos = Order.where("name = :name AND pay_type = :pay_type",
  :pay_type => params[:pay_type], :name => params[:name])
```

On myös mahdollista antaa parametrina pelkkä hash. Tällöin Rails generoi `WHERE`-lauseen, jossa avaimia käytetään kenttien niminä ja arvoja haettavina arvoina. Tällöin edellisen esimerkin voi kirjoittaa näin: (8, s. 283)

```
pos = Order.where(:name => params[:name],
  :pay_type => params[:pay_type])
```

Tai jopa näin: (8, s. 283)

```
pos = Order.where(params)
```

Viimeksi mainittua tapaa kannattaa tosin käyttää varoen, koska metodi käyttää kaikkia hashissa olevia avain-arvopareja. (8, s. 284.)

`ActiveRecord::Relation` toteuttaa useita metodeja. `First` palauttaa ensimmäisen rivin. `All` palauttaa kaikki rivit taulukkona. `ActiveRecord::Relation` tukee myös monia `Array`-luokan metodeja, kuten `each` ja `map`. Kyseiset metodit yksinkertaisesti kutsuvat ensin `all`-metodia. (8, s. 284.)

Tietokantakyselyä ei suoriteta ennen kuin jotakin edellä mainituista metodeista kutsutaan. Niin ollen kyselyä voi muokata kutsumalla muita metodeja ensin. (8, s. 285.)

Rivien ei luvata olevan missään tietyssä järjestyksessä, ellei ohjelmoija tietoisesti kutsu `order`-metodia. `Order`-metodilla voi määrittää kriteerit, jotka SQL:ssä kirjoitettaisiin `ORDER BY` -avainsanojen perään. Esimerkiksi seuraava kysely palauttaisi Daven tilaukset järjestettyinä ensisijaisesti maksutavan ja toissijaisesti toimitusajan perusteella: (8, s. 285)

```
orders = Order.where(:name => "Dave").
  order("pay_type, shipped_at DESC")
```

Palautettavien rivien määrää voi rajoittaa `limit`-metodilla. Kun `limit`-metodia käytetään, yleensä halutaan myös määrittää rivien järjestys, jotta tulokset olisivat johdonmukaisia. Seuraava esimerkki palauttaa ensimmäiset kymmenen tilausta, jotka vastaavat hakua: (8, s. 285)

```
orders = Order.where(:name => "Dave").
  order("pay_type, shipped_at DESC").limit(10)
```

`Offset`-metodilla voi määrittää, kuinka monta riviä ohitetaan. (8, s. 285.)

```
# Näkymä näyttää tilaukset sivuihin jaettuina,
# missä kukin sivu näyttää page_size tilausta kerrallaan.
# Metodi palauttaa tilaukset sivulla page_num
# (ensimmäinen sivu on 0)
def Order.find_on_page(page_num, page_size)
  order(:id).limit(page_size).offset(page_num*page_size)
end
```

`Offset`-metodia voi käyttää yhdessä `limit`-metodin kanssa, jolloin tulosten läpi käydään N riviä kerrallaan. (8, s. 285.)

Rails voi myös laskea tilastotietoja kentän arvoista. Esimerkiksi jos käytössä on tilaustaulu, voidaan laskea seuraavat tiedot: (8, s. 287)

```
average = Order.average(:amount)
max = Order.maximum(:amount)
min = Order.minimum(:amount)
total = Order.sum(:amount)
number = Order.count
```

Kaikki nämä metodit on sidottu tietokantakohtaisiin keräytymäfunktioihin, mutta niitä voi käyttää riippumatta websovelluksen käyttämän tietokannan tyypistä. (8, s. 287.)

Näitä metodeja voi käyttää yhdessä muiden metodien kanssa: (8, s. 287)

```
Order.where("amount > 20").minimum(:amount)
```

Jos tietokanta on usean prosessin tai jopa usean sovelluksen käytössä, on mahdollista, että tietokannasta haettu olio on vanhentunut: joku on voinut tallentaa tietokantaan uudempia tietoja. (8, s. 290.)

Transaktiot hoitavat ongelman jossain määrin. On kuitenkin tilanteita, joissa olio on päivitettävä käsin. ActiveRecord tekee päivittämisen helpoksi: yksinkertaisesti kutsu `reload`-metodia, ja määreiden arvot haetaan tietokannasta. (8, s. 290.)

```
stock = Market.find_by_ticker("RUBY")
loop do
  puts "Price = #{stock.price}"
  sleep 60
  stock.reload
end
```

Käytännössä `reload`ia käytetään yleensä vain yksikkötesteissä. (8, s. 290.)

Tietokannan päivittämisestä on huomattavasti vähemmän kerrottavaa kuin tietojen hakemisesta. Jos käytettävissä on ActiveRecord-olio, sen voi tallentaa tietokantaan `save`-metodilla. Jos olio on alun perin haettu tietokannasta, metodi päivittää olemassa olevan tietokantarivin: muussa tapauksessa metodi luo uuden rivin. (8, s. 290.)

```
order = Order.find(123)
order.name = "Fred"
order.save
```

ActiveRecord mahdollistaa myös olion määreiden muuttamisen ja olion tallentamisen yhdellä metodikutsulla: (8, s. 291)

```
order = Order.find(321)
order.update_attributes(:name => "Barney",
  :email => "barney@bedrock.com")
```

`Update_attributes`-metodia käytetään yleisimmin toiminnoissa, jotka liittävät lomakkeesta saatua tietoa olemassa olevaan tietokantariviin: (8, s. 291)

```
def save_after_edit
  order = Order.find(params[:id])
  if order.update_attributes(params[:order])
    redirect_to :action => :index
  else
    render :action => :edit
  end
end
```

Rivin lukemisen ja päivittämisen voi yhdistää luokkametodeilla `update` ja `update_all`. `Update` ottaa vastaan parametreina olion ID:n ja määrehashin. Se hakee rivin tietokannasta, päivittää annetut määreet, tallentaa rivin takaisin tietokantaan ja palauttaa mallioliion. (8, s. 292.)

```
order = Order.update(12, :name => "Barney",
  :email => "barney@bedrock.com")
```

`Update`-metodille voi antaa myös ID-taulukon ja määrehashtaulukon, jolloin metodi päivittää asiaan kuuluvat tietokantarivit ja palauttaa mallioliotaulukon. (8, s. 292.)

`Update_all`-metodi puolestaan mahdollistaa `UPDATE`-komennon `SET`- ja `WHERE`-lauseiden määrittämisen. Seuraava koodiesimerkki nostaa kaikkien tuotteiden, joiden nimessä on Java, hintaa kymmenen prosenttia: (8, s. 292)

```
result = Product.update_all("price = 1.1*price",
  "title LIKE '%Java%'")
```

`Update_all`-metodin paluuarvo riippuu tietokanta-ajurista. Useimmat palauttavat muuttuneiden rivien määrän. (8, s. 292.)

Aiemmin käsitellyistä `save`- ja `create`-metodeista on kaksi versiota. Versioiden välinen ero on tapa kertoa virheistä: (8, s. 292)

- `Save` palauttaa `true`n, jos olio tallentui, ja muulloin `nil`in.
- `Save!` palauttaa `true`n, jos tallennus onnistui, ja muulloin heittää poikkeuksen.
- `Create` palauttaa ActiveRecord-olion riippumatta siitä, onnistuiko tallennus. Jos ohjelmoija haluaa tietää, tallentuiko olio, hänen on tarkistettava, läpäisikö olio validoinnin (`valid?`).
- `Create!` palauttaa ActiveRecord-olion onnistuessaan ja heittää muulloin poikkeuksen.

```
if order.save
  # kaikki kunnossa
else
  # ei läpäissyt validointia
end
```

`Save`-metodia käytettäessä ohjelmoija voi päättää, tarkistetaanko metodin paluuarvo. ActiveRecordin löysyys johtuu siitä, että kehittäjät ovat olettaneet, että `save`-metodia kutsutaan toiminnossa ja näkymäkoodi näyttää kaikki virheet käyttäjälle. Monissa sovelluksissa niin myös tehdään. (8, s. 293.)

Jos olio kuitenkin tallennetaan kontekstissa, jossa on varmistettava, että kaikki virheet käsitellään ohjelmallisesti, on syytä käyttää `save!`-metodia. Metodi heittää `RecordInvalid`-poikkeuksen, jos oliota ei voi tallentaa: (8, s. 293)

```
begin
  order.save!
rescue RecordInvalid => error
  # ei läpäissyt validointia
end
```

Tietokantarivien poistamiseen voi käyttää `destroy`-metodia, joka poistaa mallioliota vastaavan rivin. Se myös jäädyttää olion, mikä estää määreiden arvon muuttamisen. (8, s. 293.)

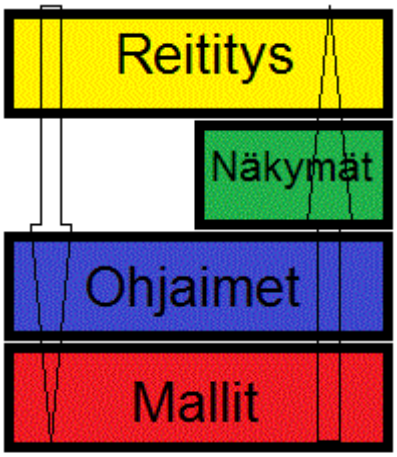
```
order = Order.find_by_name("Dave")
order.destroy
# order jäädytetty
```

On myös kaksi luokkatason tuhoamismetodia, `destroy` (joka vastaanottaa ID:n tai taulukollisen niitä) ja `destroy_all` (joka vastaanottaa ehdon). Kumpikin metodi lukee asiaan kuuluvat tietokantarivit malliolioihin ja kutsuu instanssitason `destroy`-metodia. Kumpikaan ei palauta mitään hyödyllistä. (8, s. 293.)

```
Order.destroy_all(["shipped_at < ?", 30.days.ago])
```

4.2.2 Mallit

Kuva 2 esittää Rails-pinoa. Kun Rails vastaanottaa HTTP-pyynnön, reititys määrittää, mikä ohjain käsittelee pyynnön. Ohjain hakee tietoja malleilta ja piirtää näkymän, jonka Rails lähettää käyttäjälle. Tässä aliluvussa käsittelen mallien kehittämistä.



Kuva 2. Rails-pino.

Uusi malli luodaan ajamalla seuraavanlainen komento terminaalissa:

```
rails generate model Product name:string description:text
```

Edellinen komento luo mallin nimeltään `Product`. Mallissa on kaksi määrettä: `name`, joka on lyhyt merkkijono, ja `description`, joka on pitkä merkkijono.

Komento luo myös tietokantamigraation. Migraatio on skripti, joka muuttaa tietokannan rakennetta: tässä tapauksessa luo tietokantaan `products`-taulun. Migraation voi ajaa seuraavalla komennolla:

```
rake db:migrate
```

Oletuksena mallin tietoja ei validoida ollenkaan, eli on mahdollista tallentaa malliolio riippumatta sen tiedoista, jopa ilman mitään tietoja. Usein se ei tyydytä websovelluksen kehittäjiä. Sen vuoksi ActiveRecordissa on sisäänrakennettu validointituki.

ActiveRecord määrittää useita validointiauttajia, joita voi käyttää luokkamääritysten sisällä. Käyn läpi muutaman validointiauttajan esimerkkien avulla.

```
validates :title, :description, :image_url, :presence => true
```

`:presence => true` tarkoittaa, että validaattorin on tarkistettava, että nimetyistä määreistä jokaisella on arvo eikä yksikään arvoista ole tyhjä. (8, s. 78.)

```
validates :price, :numericality =>
  {:greater_than_or_equal_to => 0.01}
```

Edellinen esimerkki saa validaattorin tarkistamaan, että hinta on luku ja vähintään 0,01. (8, s. 78.)

```
validates :title, :uniqueness => true
```

Saa validaattorin tarkistamaan, että jokaisella oliolla on eri otsikko. (8, s. 78.)

```
validates :image_url, :allow_blank => true, :format => {
  :with => %r{\.(gif|jpg|png)$}i,
  :message => "täytyy olla GIF-, JPG- tai PNG-kuvan osoite"}
```

Saa validaattorin tarkistamaan, että kuvan URL vastaa annettua säännöllistä lauseketta (8, s. 78). ActiveRecord pystyy generoimaan automaattisesti virheilmoituksen, jonka voi välittää edelleen käyttäjälle, mutta säännöllisen lausekkeen tapauksessa virheilmoitus olisi niin kryptinen, että ohjelmoijan kannattaa ohittaa se.

Kaikki paitsi yksinkertaisimmat websovellukset sisältävät useita tietokantatauluja, ja taulujen välillä on usein viittauksia (esimerkiksi ostoskoritaulussa voi olla viittauksia tuotetauluun). Katsotaan, kuinka ActiveRecord tukee sitä.

Ohjelmoijan on huolehdittava, että viittaukset voi tallentaa tietokantaan. Helpoimmin se onnistuu mallia luodessa:

```
rails generate model LineItem product_id:integer cart_id:integer
rake db:migrate
```

Sen lisäksi Railsin on tiedettävä mallien välisestä suhteesta. Yhdessä ostoskorissa voi olla monta tuoteriviä. Se määritetään näin: (8, s. 107)

```
class Cart < ActiveRecord::Base
  has_many :line_items, :dependent => :destroy
end
```

Rivin alkuosa ei juuri kaipaa selittämistä. Sen loppuosa, `:dependent => :destroy`, tarkoittaa, että tuoterivien olemassaolo riippuu vastaavasta ostoskorista. Jos ostoskori tuhotaan, myös tuoterivit on tuhottava. (8, s. 107.)

Vastaavasti tuoteriviin liittyy yksi tuote ja yksi ostoskori. Sen voi määrittää Railsille käyttämällä `belongs_to`-määritystä kahdesti `LineItem`-luokassa: (8, s. 107)

```
class LineItem < ActiveRecord::Base
  belongs_to :product
  belongs_to :cart
end
```

Mitä hyötyä määrityksistä on? Ne lisäävät malliolioihin navigointimahdollisuuksia. Koska `LineItem`-luokassa on `belongs_to`-määritys, ohjelmoija voi hakea tuoteriviä vastaavan tuotteen ja näyttää sen nimen: (8, s. 107)

```
li = LineItem.find(…)
puts "This line item is for #{li.product.title}"
```

Ja koska `Cart`-luokassa on `has_many`-määritys, ohjelmoija voi hakea tuoterivit (`ActiveRecord::Relation`-oliona) ostoskorioliosta: (8, s. 107)

```
cart = Cart.find(…)
puts "This cart has #{cart.line_items.count} line items"
```

Viimeiseksi vähän laajempi esimerkki, jossa `Product`-luokalla on monta tuoteriviä, mutta sen lisäksi luokka käyttää tuoterivejä validoinnissa. Validoinnilla estetään sellaisten tuotteiden poistaminen, joita asiakkailla on ostoskoreissa. (8, s. 107.)

```
class Product < ActiveRecord::Base
  has_many :line_items

  before_destroy :ensure_not_referenced_by_any_line_item

  # …

  private

  # varmistetaan, ettei tuotetta ole yhdessäkään ostoskorissa
  def ensure_not_referenced_by_any_line_item
    if line_items.empty?
      return true
    else
      errors.add(:base, "Line Items present")
      return false
    end
  end
end
```

Koodi määrittää, että tuotteella on monta tuoteriviä, ja toteuttaa koukkumetodin (hook) nimeltään `ensure_not_referenced_by_any_line_item`. Koukku on metodi, jota Rails kutsuu automaattisesti tietyllä hetkellä olion elinkaaren aikana. Tässä tapauksessa Rails kutsuu metodia ennen kuin yrittää tuhota tietokantarivin. Jos metodi palauttaa `falsen`, riviä ei tuhota. (8, s. 108.)

`Errors`-olio on paikka, jossa Railsin oma validaattori pitää virheilmoituksia. Virheilmoitukset voi sitoa yksittäisiin määreisiin, mutta tässä tapauksessa se sidotaan koko olioon. (8, s. 108.)

4.2.3 Näkymät

Näkymät sijaitsevat hakemistossa `app/views`. Hakemistossa on alihakemisto jokaista ohjainta kohti ja jokaisessa alihakemistossa on näkymä jokaista toimintoa kohti. Esimerkiksi `OrganizationsController`-ohjaimen `edit`-toimintoa vastaa (Hamlia käytettäessä) näkymä `app/views/organizations/edit.html.haml`. Kun ohjelmoija luo ohjaimen Railsin generaattorilla, generaattori luo vastaavat näkymät automaattisesti. Sen sijaan kun ohjelmoija lisää toimintoja jälkikäteen, näkymät on luotava käsin.

Kuva 3. App-hakemiston alihakemistoja.

Ohjelmasuomen koodissa lähes kaikki näkymät on kirjoitettu Haml-kielellä. Haml (HTML Abstraction Markup Language) on kevyt merkintäkieli, joka käännetään HTML:ksi. Hamlin tavoite on korjata monia perinteisten mallinemoottorien ongelmia ja tehdä sivujen lähdekoodista mahdollisimman eleganttia. Hamlilla voi korvata PHP:n, ERB:n, ASP:n ja muita webmallinejärjestelmiä. Toisin kuin edellä mainituissa järjestelmissä, Hamlissa kehittäjän ei tarvitse kirjoittaa XHTML-syntaksia, koska Haml on itsessään kuvaus XHTML:stä. (11.)

Hamlissa Ruby-koodia voi sijoittaa näkymään yhtäsuuruusmerkin avulla. Haml-tulkki evaluoi koodinpätkän ja sijoittaa paluuarvon dokumenttiin. Esimerkiksi seuraava koodiesimerkki (12)

```
%p
  = ["hi", "there", "reader!"].join " "
  = "yo"
```

tuottaa seuraavaa HTML:ää: (12)

```
<p>
  hi there reader!
  yo
</p>
```

Yhtäsuuruusmerkin voi kirjoittaa myös tunnisteen perään, jolloin Haml-tulkki sijoittaa koodin tunnisteen sisään. Esimerkiksi seuraava koodiesimerkki (12)

```
%p= "hello"
```

tuottaa seuraavaa HTML:ää: (12)

```
<p>hello</p>
```

Ruby-koodirivin voi jakaa usealle riville, jos kaikki rivit viimeistä lukuun ottamatta loppuvat pilkkuun. (12.)

```
= link_to_remote "Add to cart",
  :url => {:action => "add", :id => product.id},
  :update => {:success => "cart", :failure => "error"}
```

Ruby-koodia voi suorittaa yhdysmerkin avulla. Silloin Haml-tulkki evaluoi koodinpätkän, muttei sijoita paluuarvoa minnekään. Esimerkiksi seuraava koodiesimerkki (12)

```
- foo = "hello"
- foo << " there"
- foo << " you!"
%p= foo
```

tuottaa seuraavaa HTML:ää: (12)

```
<p>hello there you!</p>
```

Ruby-koodirivin voi jakaa usealle riville samalla tavalla kuin yhtäsuuruusmerkkiä käytettäessä. (12.)

```
- links = {:home => "/",
  :docs => "/docs",
  :about => "/about"}
```

Ruby-lohkoja ei tarvitse sulkea eksplisiittisesti. Haml-tulkki avaa lohkon automaattisesti, kun sisennystä lisätään, ja päättää sen, kun sisennystä vähennetään (paitsi jos rivillä on `else`-lause tai jotakin vastaavaa). Esimerkiksi seuraava koodiesimerkki (12)

```
- (42...47).each do |i|
  %p= i
%p See, I can count!
```

tuottaa seuraavaa HTML:ää: (12)

```
<p>42</p>
<p>43</p>
<p>44</p>
<p>45</p>
<p>46</p>
<p>See, I can count!</p>
```

Ruby-koodia voi myös interpoloida tekstiin `#{}`-operaattorilla samoin kuin Ruby-merkkijonoissa. Esimerkiksi (12)

```
%p This is #{h quality} cake!
```

tarkoittaa samaa kuin (12)

```
%p= "This is #{h quality} cake!"
```

ja voi tuottaa esimerkiksi tällaista HTML:ää: (12)

```
<p>This is scrumptious cake!</p>
```

Kenoviivalla voi estää `#{}`-merkkijonon tulkinnan, mutta kenoviiva ei toimi pakomerkkinä missään muualla. Esimerkiksi seuraava koodiesimerkki (12)

```
%p
  Look at \\#{h word} lack of backslash: \#{foo}
  And yon presence thereof: \{foo}
```

voi tuottaa esimerkiksi tällaista HTML:ää: (12)

```
<p>
  Look at \yon lack of backslash: #{foo}
  And yon presence thereof: \{foo}
</p>
```

Interpolaatiota voi käyttää myös Haml-suodattimien sisällä. Esimerkiksi seuraava koodiesimerkki (12)

```
:javascript
  $(document).ready(function() {
    alert(#{@message.to_json});
  });
```

voi tuottaa esimerkiksi tällaista HTML:ää: (12)

```
<script type="text/javascript">
  //<![CDATA[
    $(document).ready(function() {
      alert("Hi there!");
    });
  //]]>
</script>
```

Haml-tulkin voi myös saada korvaamaan erikoismerkit Ruby-koodin paluuarvossa HTML-entiteeteillä &=-operaattorin avulla. Esimerkiksi seuraava koodiesimerkki (12)

```
&= "I like cheese & crackers"
```

tuottaa seuraavaa HTML:ää: (12)

```
I like cheese &amp; crackers
```

&-merkkiä voi käyttää myös yksinään, jolloin se saa tulkin korvaamaan erikoismerkit entiteeteillä #{}-interpolaatiossa. Esimerkiksi seuraava koodiesimerkki (12)

```
& I like #{"cheese & crackers"}
```

tuottaa seuraavaa HTML:ää: (12)

```
I like cheese &amp; crackers
```

Useimmissa projekteissa suurta osaa näkymäkoodista halutaan käyttää kaikissa näkymissä. Koodia ei kannata kopioida jokaiseen näkymään, koska jos sitä halutaan muuttaa, muutos on tehtävä pahimmassa tapauksessa kymmenissä näkymissä.

Sen vuoksi Rails mahdollistaa näkymäkoodin jakamisen asettelun (layout) avulla. Asettelu on näkymä, jonka sisään kaikki muut näkymät piirretään. Hamlia käytettäessä asettelun polku on `app/views/layouts/application.html.haml`. Ohjelmasuomen asettelu on seuraava:

```
!!! 5
%html(lang="en")
  %head
    %meta(charset="utf-8")
    %meta(name="viewport"
      content="width=device-width, initial-scale=1.0")
    %title= content_for?(:title) ? yield(:title) : "Ohjelmasuomi"
    = csrf_meta_tags
    / Le HTML5 shim, for IE6-8 support of HTML elements
    /[if lt IE 9]
    = javascript_include_tag "html5shiv.js"
    = stylesheet_link_tag "application", :media => "all"
    %link(href="images/favicon.ico" rel="shortcut icon")
    %link(href="images/apple-touch-icon.png"
      rel="apple-touch-icon")
    %link(href="images/apple-touch-icon-72x72.png"
      rel="apple-touch-icon" sizes="72x72")
    %link(href="images/apple-touch-icon-114x114.png"
      rel="apple-touch-icon" sizes="114x114")

  %body
    %header.container-fluid
      .row-fluid
        .span24
          =render 'layouts/header'
    %section.container-fluid
      .row-fluid
        .span19.content#content
          -if flash[:notice]
            %div.alert.alert-info.fade.in
              %br
              =flash[:notice]
          -if flash[:alert]
            %div.alert.alert-error.fade.in
              %br
              =flash[:alert]
          = yield
        .span5.sidebar
          =render 'layouts/sidebar'

    %footer.container-fluid
      .row-fluid
        =render 'layouts/footer'
```

```
/
  Javascripts
  \==============================================
/ Placed at the end of the document so the pages load faster
= javascript_include_tag "application"
```

Varsinainen näkymä piirretään riville, jolla lukee `= yield`.

`Csrf_meta_tags` on metodi, joka tuottaa `<meta>`-tunnisteita osana CSRF-hyökkäysten estämistä. Sen tuottamat tunnisteet ovat osittain satunnaisgeneroituja, mutta suunnilleen tällaisia:

```
<meta content="authenticity_token" name="csrf-param">
<meta content="JdDWlNime8+wznnMPDczA3Y1bHNwJBzPk/682UHf7q4="
name="csrf-token">
```

`Javascript_include_tag` on metodi, joka tuottaa tunnisteen, joka sisällyttää halutun JavaScript-tiedoston dokumenttiin. `Javascript_include_tag "application"` sisällyttää dokumenttiin tiedoston `app/assets/javascripts/application.js`. Kehitysympäristössä se tuottaa seuraavan tunnisteen:

```
<script src="/assets/application.js"
type="text/javascript"></script>
```

Vastaavasti `stylesheet_link_tag` tuottaa tunnisteen, joka sisällyttää halutun tyyliarkin dokumenttiin. `Stylesheet_link_tag "application"` sisällyttää dokumenttiin tiedoston `app/assets/stylesheets/application.css`. Kehitysympäristössä se tuottaa seuraavan tunnisteen:

```
<link href="/assets/application.css" rel="stylesheet"
type="text/css" />
```

Suositeltu tapa lisätä näkymään linkki on `link_to`-metodi.

```
= link_to "Hylkää kaikki", hide_organization_path(@organization)
```

Metodin ensimmäinen parametri on linkin teksti ja toinen URL, johon linkki osoittaa. Rails generoi jokaiselle nimetylle reitille (named route) metodin, joka palauttaa kyseisen reitin polun. Esimerkiksi Ohjelmasuomessa on reitti `hide_organization`, joten Rails on generoinut metodin `hide_organization_path`.

Jotenkin on myös määritettävä, mikä organisaatio piilotetaan, jos ylläpitäjä seuraa yllä olevaa linkkiä. Tässä tapauksessa organisaation ID määritetään polussa: polku on muotoa `organization/:id/hide`.

Jotta `hide_organization_path` voi tuottaa polun, sen on tiedettävä `:id`-parametrin arvo. Sen vuoksi edellisessä koodiesimerkissä metodille annetaan viite organisaatioon. Metodi tarkistaa parametrin tyypin: tässä tapauksessa se on ActiveRecord-malli, joten metodi sisällyttää polkuun mallin ID:n. (Esimerkiksi luvut ja merkkijonot sijoitetaan polkuihin sellaisenaan.)

Edellinen koodiesimerkki voi tuottaa esimerkiksi tällaista HTML:ää:

```
<a href="/organization/11/hide">Hylkää kaikki</a>
```

`Link_to`-metodilla on myös kolmas, mutta vapaaehtoinen parametri: hash, joka sisältää asetuksia. Voi herätä kysymys, mitä asetuksia on saatavilla.

Suositeltu tapa ottaa se selville on mennä Railsin viralliselle API-sivustolle (13) ja kirjoittaa `link_to` sivun vasemmassa yläkulmassa olevaan hakulaatikkoon. Samalla tavalla voi selvittää kaikkien Railsin metodien asetukset.

Jos malli käyttää RESTful-reittejä, tärkeimmät käytettävissä olevat polut ovat

Toiminto	Koodi	URL
Luettelo organisaatioista	`organizations_path`	/organizations
Lomake, jolla voi luoda uuden organisaation	`new_organization_path`	/organizations/new
Näytä organisaatio	`organization`	/organizations/11
Lomake, jolla voi muokata organisaatiota	`edit_organization_path(organization)`	/organizations/11/edit
Poista organisaatio	`organization, :method => :delete`	/organizations/11

Esimerkeissä `organization` on olemassa oleva `Organization`-olio ja koodi tarkoittaa, mitä kirjoitetaan `link_to`-metodikutsuun. Esimerkiksi organisaation poistamislinkki tehdään näin:

```
-# organization on Organization-olio
= link_to "Poista organisaatio", organization, :method => :delete
```

Seuraavana on pätkä Ohjelmasuomen sivulta `app/views/search/index_orgs.html.haml`. (Olen poistanut pari riviä, jotka vaikeuttaisivat esimerkin ymmärtämistä.)

```
%h2.box-title Tulokset
- @result.each do |org|
  - if org.profile.publish
    .box.straight-corners.white.no-border
      .row-fluid
        .span18
           .image.logo-header-show.no-color
            -if org.profile.logo_header
              = image_tag org.profile.logo_header.thumb('680x70').url
            - else
              %h3= org.profile.title
          .description
            = truncate org.profile.description, :length => 800
```

`@result` on `ActiveRecord::Relation`-olio. `Each`-metodi on iteraattori, joka antaa joka kierroksella `Organization`-tyyppisen alkion parametrina. Lohko antaa parametrille nimen `org`. Lohkon sisällä tarkistetaan, onko organisaatiolla otsikkologo:

```
-if org.profile.logo_header
```

Jos on, kyseinen otsikkologo piirretään näkymään:

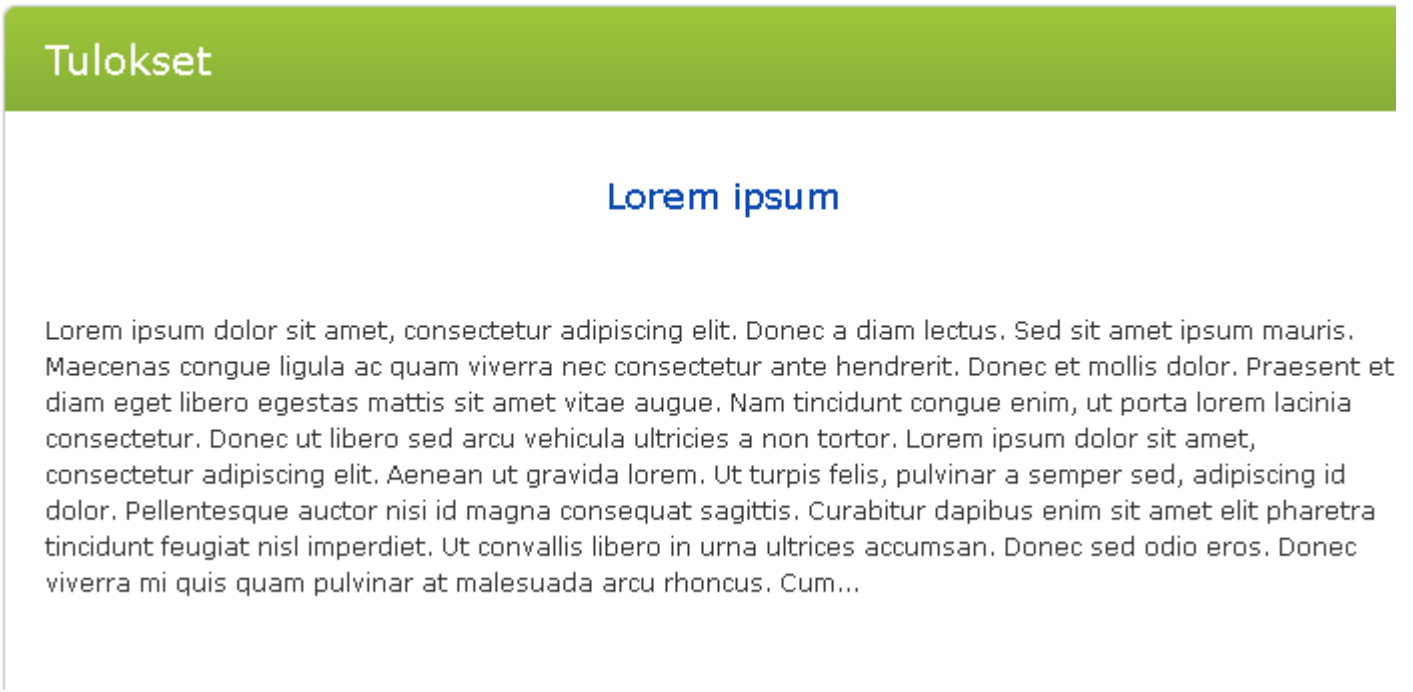
```
= image_tag org.profile.logo_header.thumb('680x70').url
```

Muussa tapauksessa piirretään organisaation nimi:

```
- else
  %h3= org.profile.title
```

Näkyvissä oleva koodin osa piirtää myös organisaation kuvauksen.

Lopputulos näyttää tältä:



Kuva 4. Osa näkymästä `app/views/search/index_orgs.html.haml`.

4.2.4 Ohjaimet

Uusi ohjain luodaan ajamalla seuraavanlainen komento terminaalissa:

```
rails generate controller Store index
```

Edellinen komento luo ohjaimen nimeltään `StoreController`, jolla on vain yksi toiminto: `index`. Komento luo myös reitin `index`-toimintoon (`store_path`, /store) ja tarvittavan näkymän (`app/views/store/index.html.haml`).

Jokainen ohjain on Ruby-luokka ja jokainen toiminto on metodi. Esimerkiksi `index`-toiminto on `StoreController`-luokan `index`-metodi.

Yleisesti ottaen näkymissä kannattaa pitää mahdollisimman vähän sovelluslogiikkaa. Logiikan on parasta olla ohjaimissa tai vielä mieluummin malleissa. Erinomainen esimerkki on edellisellä sivulla mainittu näkymä `app/views/search/index_orgs.html.haml`. Haku on niin monimutkainen ominaisuus, että lähes kaikki koodi on mallissa.

`SearchController` näyttää tältä:

```
class SearchController < ApplicationController
  def index

    # onko yritys- vai palveluhaku
    if params[:type] == 'organisation'
      @result = Search.search_org(params)
      render 'index_orgs'
    elsif params[:type] == 'service'
      @result = Search.search_serv(params)
      render 'index_services'
    # toimiessaan elseen ei päästä
    else
      render 'index_orgs'
    end
  end
end
```

Kuten näkyy, ohjain delegoi itse hakemisen mallille. Se myös välittää tuloksen näkymälle. Railsissa näkymä pääsee käsiksi ohjaimen instanssimuuttujiin, ja siksi näkymälle välitettävien tietojen täytyy olla instanssimuuttujissa eikä paikallisissa muuttujissa.

Ohjaimessa käytetään myös `params`-oliota. Se on hash, joka sisältää parametrit.

Jos ohjelmoija haluaa palauttaa HTML:n sijaan XML:ää tai JSON:ia, se onnistuu seuraavasti: (14)

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html
      format.xml {render :xml => @users}
      format.json {render :json => @users}
    end
  end
end
```

Esimerkissä `html`-metodille ei annettu lohkoa. Se nimittäin on tarpeetonta: Rails tietää, että jokaiseen HTTP-pyyntöön on vastattava, ja tarkistaa toiminnon lopussa, onko pyyntöön vastattu. Ellei ole, Rails piirtää automaattisesti vastaavan nimisen näkymän (yllä olevassa esimerkissä `app/views/users/index.html.haml`, jos Hamlia käytetään). Se on yleisin tapa saada näkymä piirrettyä. (8, s. 322.)

On toimenpiteitä, joita on tehtävä useissa toiminnoissa tai jopa useissa ohjaimissa. Hyvä esimerkki on tarkistus, että käyttäjä on ylläpitäjä. Sellainen tarkistus on tehtävä kaikissa vaarallisissa toiminnoissa koko sovelluksessa, mutta ei muissa.

Koodin kopioinnilta vältytään käyttämällä Railsin suodattimia (filters). Suodatin sieppaa toimintojen kutsut ja tekee jotakin ennen kuin toimintoa kutsutaan ja/tai kun toiminto on suoritettu (8, s. 200). Tähän tilanteeseen sopii ensin-suodatin (before filter), joka suoritetaan ennen toiminnon kutsumista.

Koska suodatinta tarvitaan niin monessa paikassa, Ohjelmasuomen koodissa se on pantu `ApplicationController`-luokkaan, joka on kaikkien ohjainten yliluokka.

```
class ApplicationController < ActionController::Base
  # …

  private

  def authenticate_admin_user!

     unless authenticate_user! && current_user.has_role?(:admin)
         flash[:alert]  = "Permission denied"
         redirect_to :root
     end
  end
```

Suodatinta käytetään `AdminController`-luokassa, joka on ylläpitotoimintoja tarjoavien ohjainten yliluokka, tähän tapaan:

```
class Administration::AdminController < ApplicationController
   before_filter :authenticate_admin_user!

    def index
    end

end
```

4.2.5 Reititys

Railsissa on vielä yksi pääkomponentti, jota en ole käsitellyt. Jos ylläpitäjä painaa linkkiä, joka johtaa vaikkapa osoitteeseen ohjelmasuomi.fi/organizations/11/hide, mistä Rails arvaa, mikä toiminto on suoritettava?

Vastaus on reititys. Tiedosto `config/routes.rb` sitoo URL:t toimintoihin. Aina kun palvelin vastaanottaa HTTP-pyynnön, se etsii tiedostosta reitin, joka vastaa annettua URL:ää ja suorittaa reitin määrittämän toiminnon.

Suositeltu tapa luoda reittejä on `resources`-metodi, joka luo RESTful-reittejä. Esimerkki: (8, s. 310)

```
resources :products
```

Hyvä tapa saada selville, mitä yllä oleva rivi oikeastaan tekee, on suorittaa komento `rake routes` terminaalissa: (8, s. 310)

```
products    GET         /products(.:format)
            {:action=>"index", :controller=>"products"}
            POST        /products(.:format)
            {:action=>"create", :controller=>"products"}
new_product GET         /products/new(.:format)
            {:action=>"new", :controller=>"products"}
edit_productGET         /products/:id/edit(.:format)
            {:action=>"edit", :controller=>"products"}
product     GET         /products/:id(.:format)
            {:action=>"show", :controller=>"products"}
            PUT         /products/:id(.:format)
            {:action=>"update", :controller=>"products"}
            DELETE      /products/:id(.:format)
            {:action=>"destroy", :controller=>"products"}
```

`Rake routes` tulostaa kaikki reitit taulukkomuodossa. Taulukon sarakkeet ovat reitin nimi (vapaaehtoinen), HTTP-verbi, polku ja toiminto. (8, s. 310.)

Polussa sulut tarkoittavat vapaaehtoista polun osaa (ts. polku toimii, vaikka suluissa olevat osat puuttuisivat URL:stä). Kaksoispisteet puolestaan määrittävät parametreja, jotka reititin tulee sijoittamaan `params`-hashiin toiminnon käyttöön. (8, s. 310.)

Kuten näkyy, `resources` luo oletusasetuksilla seitsemän reittiä. Reittien merkitykset ovat seuraavat: (8, s. 311)

- `index`

  Palauttaa luettelon resursseista. (8, s. 311.)

- `create`

  Luo uuden resurssin POST-pyynnössä olevasta datasta ja tallentaa resurssin tietokantaan. (8, s. 311.)

- `new`

  Luo ja palauttaa uuden resurssin. Resurssia ei tallenneta tietokantaan. Voidaan ajatella, että `new`-toiminto luo tyhjän lomakkeen asiakkaan (client) täytettäväksi. (8, s. 311.)

- `show`

  Palauttaa resurssin, jonka ID on `params[:id]`, sisällön. (8, s. 311.)

- `update`

  Päivittää resurssin, jonka ID on `params[:id]`, sisällön pyyntöön sidotulla datalla. (8, s. 311.)

- `edit`

  Palauttaa resurssin, jonka ID on `params[:id]`, sisällön muodossa, joka sopii muokkaamiseen. (8, s. 311.)

- `destroy`

  Tuhoaa resurssin, jonka ID on `params[:id]`. (8, s. 311.)

`Create`, `show`, `update` ja `destroy` voivat kuulostaa tutuilta – ne ovat CRUD-operaatiot. Rails lisää `index`-toiminnon, joka palauttaa luettelon resursseista, ja kaksi toimintoa, jotka palauttavat resursseja muodossa, joka sopii muokkaamiseen. (8, s. 311.)

Ellei ohjelmoija halua kaikkia seitsemää reittiä, niitä voi rajoittaa `:only`- tai `:except`-asetuksella: (8, s. 311)

```
resources :comments, :except => [:update, :destroy]
```

Osa `resources`-metodin luomista reiteistä on nimettyjä, mikä mahdollistaa polkumetodien (kuten `products_url` ja `edit_product_url`) käytön. (8, s. 311.)

Ohjelmoijan ei ole mikään pakko tyytyä valmiisiin `resources`-metodin luomiin reitteihin. Ylimääräisiä reittejä voi luoda antamalla `resources`-metodille lohkon:

```
resources :products do
  get :who_bought, :on => :member
end
```

Edellinen esimerkki tarkoittaa: ”Haluan toiminnon nimeltään `who_bought`, jota kutsutaan GET-verbillä. Toiminto voidaan suorittaa kullekin tuotekokoelman (collection) alkiolle.” (8, s. 315.)

Toinen sallittu arvo `:on`-asetukselle on `:collection`, joka tarkoittaa, että toiminnon voi suorittaa vain koko kokoelmalle (ts. toiminto ei ota vastaan tuotteen ID:tä). (8, s. 315.)

Reitti voi myös uudelleen ohjata käyttäjän toiseen polkuun. (15.)

```
match "/stories" => redirect("/posts")
```

Polussa olevia parametreja voi käyttää kohdepolun osina: (15)

```
match "/stories/:name" => redirect("/posts/%{name}")
```

`Redirect`-metodille voi antaa myös lohkon, jolle annetaan parametrit ja pyyntö: (15)

```
match "/stories/:name" => redirect {|params|
  "/posts/#{params[:name].pluralize}"}

match "/stories" => redirect {|p, req| "/posts/#{req.subdomain}"}
```

Uudelleenohjaus on toteutettu HTTP-statuskoodilla 301 Moved Permanently. Jotkin selaimet ja välityspalvelimet tallentavat 301-uudelleenohjaukset välimuistiin, joten jotkin käyttäjät eivät enää pääse vanhalle sivulle, jos uudelleenohjaus poistetaan myöhemmin. (15.)

Ellei domainia määritetä kohdeosoitteessa, Rails käyttää pyynnön tietoja (15). Esimerkiksi jos uudelleenohjaus ohjaa käyttäjän sivulle /uutiset ja pyynnön `Hostname`-header on www.ohjelmasuomi.fi, Rails uudelleen ohjaa käyttäjän sivulle http://www.ohjelmasuomi.fi/uutiset.

Sivuston juurta vastaavan reitin voi luoda `root`-metodilla: (15)

```
root :to => 'pages#main'
```

Merkkijonossa risuaitaa edeltävä osa on ohjaimen nimi (merkkikoosta ei välitetä) ja risuaitaa seuraava osa toiminnon nimi.

Juurireitti vaikuttaa vain, jos tiedosto `public/index.html` on poistettu. Suorituskyvyn vuoksi juurireitti kannattaa panna `routes.rb`-tiedoston alkuun. (15.)

Reitin voi nimetä käyttämällä `:as`-asetusta:

```
resources :products do
  get :who_bought, :on => :member, :as => :who_bought
end
```

5 RSpec-testityökalu

Ohjelmasuomen yksikkötestit on toteutettu RSpec-testityökalulla. Koska testattavuus on yksi ylläpidettävyyden komponentti, RSpecillä on suuri vaikutus jatkokehitettävyyteen.

RSpec (16) on testityökalu Ruby-ohjelmointikielelle. RSpecin virallinen tavoite on tehdä testivetoisesta kehityksestä (TDD, Test-driven development) tuottavaa ja hauskaa. RSpecin ominaisuuksia:

- ominaisuuskylläinen komentoriviohjelma (`rspec`-komento)
- tekstimuotoiset esimerkkien ja ryhmien kuvaukset (rspec-core)
- joustava ja räätälöitävissä oleva raportointi

- laajennettava odotuskieli (expectation language) (rspec-expectations)
- sisäänrakennettu mockaus/stubbaus-ohjelmistokehys (rspec-mocks)

RSpec mahdollistaa selkeät, tiiviit ja luettavat yksikkötestit. RSpecin pahimmalla kilpailijalla, Test::Unitilla, kirjoitetut testit näyttävät tältä: (17)

```
class StackTest < Test::Unit
  def setup
    @stack = Stack.new
  end

  def test_new_is_empty
    assert_equal true,
      @stack.empty?
  end

  def test_new_has_size_0
    assert_equal 0,
      @stack.size
  end
end
```

Vastaavat RSpec-testit: (17)

```
describe Stack do
  context "when new" do
    it {should be_empty}
    it {should have(0).items}
  end
end
```

Lisäksi RSpec-testit toimivat jonkinlaisena dokumentaationa. Kun yllä olevat testit ajetaan, RSpec tulostaa tällaisen yhteenvedon: (17)

```
Stack
  when new
    should be empty
    should have 0 items

Finished in 0.00089 seconds
2 examples, 0 failures
```

6 Näkymissä käytetyt kielet ja kirjastot

Tämä luku käsittelee kieliä ja kirjastoja, joita Ohjelmasuomen näkymissä käytetään.

6.1 Sass-tyylitiedostokieli

Sass (Syntactically Awesome Stylesheets) on kevyt tyylitiedostokieli, joka käännetään CSS:ksi. Hamlia ja Sassia kehittävät samat ihmiset. (18.)

Sassilla on kaksi syntaksia. Alkuperäinen syntaksi, jota kutsutaan sisennyssyntaksiksi (the indented syntax) muistuttaa Hamlia. Se käyttää sisennystä koodilohkojen erotteluun ja rivinvaihtoja sääntöjen erotteluun. Uudempi syntaksi, nimeltään SCSS, käyttää CSS:ää muistuttavaa lohkomuotoilua. Se käyttää aaltosulkuja koodilohkojen erotteluun ja puolipisteitä sääntöjen erotteluun. Käytäntö on antaa sisennyssyntaksia käyttäville tiedostoille .sass- ja SCSS-tiedostoille .scss-tiedostopääte. (18.)

CSS3 koostuu valitsimista (selector) ja pseudovalitsimista (pseudo-selector), jotka ryhmittelevät sääntöjä. Sass laajentaa CSS:ää tarjoamalla mekanismeja, jotka ovat käytettävissä useimmissa perinteisissä ohjelmointikielissä, mutta eivät CSS3:ssa. Sass-tulkki kääntää Sassia CSS:ksi. Vaihtoehtoisesti Sass-tulkki voi valvoa .sass- tai .scss-tiedostoa ja kääntää sen aina, kun ohjelmoija tallentaa sen. (18.)

Sassin virallinen toteutus on avointa lähdekoodia ja kirjoitettu Ruby-kielellä. Muitakin toteutuksia on olemassa, kuten PHP-toteutus Drupalia varten. Sassin sisennyssyntaksi on metakieli. SCSS puolestaan on sisäkkäinen metakieli (nested metalanguage), koska validi CSS on validia SCSS:ää ja CSS:n semantiikka pysyy samana, kun se tulkitaan SCSS:nä. (18.)

Sassin mekanismeja ovat muuttujat, sisäkkäisyys, moduulit ja valitsimien perintä. (18.)

6.2 jQuery-kirjasto

JQuery on useita selaimia tukeva JavaScript-kirjasto, jonka tarkoitus on helpottaa asiakaspuolen skriptausta (client-side scripting). John Resig julkaisi jQueryn tammikuussa 2006. Nykyään jQueryä kehittää kokonainen tiimi, jota johtaa Dave Methvin. JQuery on tämän hetken yleisin JavaScript-kirjasto, jota käyttää yli 55 % maailman 10 000 suurimmasta sivustosta. (19.)

JQuery on avointa lähdekoodia ja MIT-lisensoitu. JQueryn syntaksin on tarkoitus helpottaa dokumentissa navigointia, DOM-elementtien valintaa, animointia, tapahtumien käsittelyä ja Ajax-sovellusten kehittämistä. JQueryn päälle voi myös luoda plugineja. Sen ansiosta kehittäjät voivat luoda abstraktioita matalan tason vuorovaikutusta ja animaatiota varten, kehittyneitä efektejä ja korkean tason käyttöliittymäkomponentteja. JQueryn modulaarisuus mahdollistaa ominaisuuskylläisten dynaamisten websivujen ja websovellusten kehittämisen. (19.)

JQueryn ominaisuuksia: (19)

- DOM-elementtien valinta Sizzlen avulla
- DOMin läpikäynti ja muokkaaminen
- CSS-valitsimiin perustuva DOMin manipulointi
- tapahtumat
- efektit ja animaatiot
- Ajax
- laajennettavuus plugineilla
- työkaluja, kuten selaimen tukemien ominaisuuksien tunnistaminen
- fallbackeja funktioille, joita vanhat selaimet eivät tue
- tukee useita selaimia.

7 Havaintoja

Suurin osa muutoksista, joita minua on pyydetty tekemään, ovat olleet pieniä käyttöliittymän muutoksia. Niin ollen tekemäni muutokset ovat kohdistuneet pääosin näkymiin, tyyliarkkeihin ja skripteihin.

Pienten muutosten tapauksessa tavallistakin suurempi osa kehitysajasta menee oikeiden tiedostojen etsimiseen. Rails sekä helpottaa että vaikeuttaa tiedostojen etsimistä. Etsinnän helpottuminen johtuu siitä, että Rails pakottaa noudattamaan Model View Controller -suunnittelumallia: kaikki mallit ovat hakemistossa `app/models`, näkymät hakemistossa `app/views` ja ohjaimet hakemistossa `app/controllers`.

Toisaalta jotkin Railsin ominaisuudet vaikeuttavat oikeiden tiedostojen etsimistä. Seuraavana on pätkä skriptistä `app/assets/javascripts/organizations.js.coffee`:

```
$(".upload-link").on "ajax:success", (event, data) ->
  $("#imagemodal").html(data)
  $("#imagemodal").modal("show")
  $("form.edit_profile").on "ajax:success", (event, data) ->
    $("#crop_preview").html(data)
    # Inform the script that it has been joined to DOM
    window.cropThumbnailsLoaded()
    $("#crop_form").on "ajax:success",
      (event, data, status, xhr) ->
      $("#imagemodal").modal("hide")
      img_id = "#" +
        xhr.getResponseHeader("X-Image-Id").URLDecode()
      flash_msg = xhr.getResponseHeader("X-Flash").URLDecode()
      $(img_id).html(data)
      $restInPlaceElements = $(img_id).find(".rest-in-place")
      $restInPlaceElements.restInPlace()
      $restInPlaceElements.activateRestInPlaceHooks()
      flash_success("#homepage_flash", flash_msg)
    $("#crop_form").on "ajax:error", (event, data) ->
      console.log("Ajax error!"+data.message)
  $("[data-dismiss=modal]").click ->
    $.ajax({
      url: $("#auto_crop_path").val()
      type: "POST"
    })
```

Skripti hyödyntää Railsin huomiota herättämätöntä JavaScriptiä. Kun lomakkeen tai linkin luo `:remote => true` -asetuksella,

```
= link_to edit_image_path(logoname,
  :organization => @organization),
  :remote => true,
  "data-type" => :html,
  :class => "upload-link image-container" do
    # …
```

huomiota herättämätön JavaScript-ajuri tekee Ajax-pyynnön linkin seuraamisen tai lomakkeen lähettämisen sijaan.

JQuery-ajuri laukaisee tapahtuman nimeltään `ajax:success`, kun Ajax-pyyntö on valmis. Tällöin edellisellä sivulla olevaa tapahtuman käsittelijää kutsutaan. Funktion ensimmäinen parametri on jQueryn tapahtumaolio, toinen parametri palvelimen lähettämä data (HTML:ää, JSONia, merkkijono tai vastaavaa), kolmas parametri statuskoodi ja neljäs jQueryn Ajax-pyyntöolio.

Palvelimen lähettämää dataa käytetään tällä tavalla:

```
$("#imagemodal").html(data)
```

Data on siis HTML:ää, joka yhdistetään DOMiin. Hieman alempana skripti rekisteröi toisen tapahtuman käsittelijän:

```
$("form.edit_profile").on "ajax:success", (event, data) ->
```

Tässä vaiheessa on tiedettävä, millaista HTML:ää palvelin palautti. Ensimmäiseksi on haettava tekstiä `upload-link` hakemistosta `app/views`, jotta tiedämme, mihin elementteihin alkuperäinen callback on rekisteröity.

Teksti löytyy vain yhdestä tiedostosta: `app/views/organizations/_logo.html.haml`. Osuma on ylempänä tällä sivulla. Elementti on linkki, jonka kohde on `edit_image_path(logoname, :organization => @organization)`. Seuraavaksi on selvitettävä, mihin reittiin se on sidottu.

`rake routes | less`:

```
edit_image  GET         profiles/edit_image/:id
            {:action=>"edit_image", :controller=>"profiles"}
```

Toiminto on `profiles#edit_image`. Toiminnon lähdekoodi:

```
def edit_image
  if params.has_key?(:organization) &&
    current_user.has_role?(:admin)
    @organization = Organization.find(params[:organization])
  else
    @organization = current_user.organization
  end
  @profile = @organization.profile

  #if @profile.update_attributes(params[:profile])
  #   flash[:notice] = "Tiedot päivitetty"
  #end

  respond_to do |format|
    # TODO: refactor to use edit_image/:id
    @image_name = params[:id]
    # aspect rate for crop tool
    if (@image_name =~ /logo_header/)
      @aspectrate = 680/70
    else
      @aspectrate = 1
    end
    # render crop partial
    format.html { render :partial => "imagemodal" }
  end
end
```

Toiminto piirtää siis `imagemodal`-partiaalin, jonka polku on `app/views/profiles/_imagemodal.html.haml`. Partiaalin lähdekoodi:

```
.modal-header
  %button.close{:data => {:dismiss => :modal}}
    x
  %h3 Lataa kuva sivuille #{params[:id]}
.modal-body
  = form_for @profile, :url => update_image_path(params[:id]),
  :html => { :multipart => true, "data-type" => :html },
  :remote => true do |f|
    = f.file_field params[:id]
    = hidden_field :image_id, params[:id]
    = f.hidden_field :id
    %br
    = f.submit 'Lataa kuva järjestelmään',
```

```
      :class => "btn btn-primary"
  .alert.alert-error.hide#crop_alert
  #crop_preview
.modal-footer
  -# this is specified in crop partial
  -#= link_to_function "Rajaa",
    '$("#crop_submit_form").submit()',
    :class => "btn btn-primary disabled" , :id => "crop_button"
  = link_to "Sulje", "#", :class => "btn",
    "data-dismiss" => "modal"
```

Palataan edellä olevaan skriptiin:

```
$("form.edit_profile").on "ajax:success", (event, data) ->
```

Missään ei ole elementtiä, jonka luokka on `edit_profile`. Eikö yllä oleva rivi siis tee-kään mitään?

Kyllä se tekee. Partiaalissa nimittäin on seuraava osa:

```
= form_for @profile, :url => update_image_path(params[:id]),
  :html => { :multipart => true, "data-type" => :html },
  :remote => true do |f|
```

Kun `@profile` on uusi olio (ei tallennettu tietokantaan), Rails antaa lomakkeelle auto-maattisesti luokan `new_profile`. Ja kun `@profile` on olemassa oleva olio, Rails antaa lomakkeelle luokan `edit_profile`. Kun en tiennyt sitä, skriptin tulkitseminen tuotti mi-nulle suuria vaikeuksia.

Skripti rekisteröi siis tapahtuman käsittelijän, jota kutsutaan, kun yllä oleva lomake on lähetetty ja palvelin on vastannut HTTP-pyyntöön. Katsotaan, mikä on palvelimen vas-taus tällä kertaa.

`rake routes | less`:

```
update_imageGET         profiles/update_image/:id
            {:action=>"update_image", :controller=>"profiles"}
```

`Update_image`-toiminto samassa ohjaimessa. Lähdekoodi:

```
  def update_image
    if current_user.has_role?(:admin)
      @profile = Profile.find(params[:profile][:id])
```

```
    else
      @profile = current_user.organization.profile
    end
    @id = params[:id]
    @image_name = @id
    @object = @profile
    if @id == "logo_header"
      @aspectratio = 680.0/70.0
    else
      @aspectratio = 1
    end

    if @profile.update_attributes(params[:profile])
      @image = @profile.send(@id)
      render :partial => "shared/crop_thumbnails", :locals =>
{ :image_id => @image }
    else
      render :partial => "shared/errors", :locals => { :resource
=> @profile }
    end
  end
```

Jos profiilin päivittäminen onnistuu, toiminto piirtää partiaalin `app/views/shared/_crop_thumbnails.html.haml`.

Palataan taas edellä olevaan skriptiin.

```
$("#crop_preview").html(data)
```

Skripti siis yhdistää DOMiin `crop_thumbnails`-partiaalin. Partiaalin tärkein osa on sen lopussa:

```
= form_for @object, :url => {:action => :crop_image, :id => @id},
  :html => { "data-type" => :html, :id =>  "crop_form"},
  :remote => true do |f|
  - for attribute in [:x, :y, :width, :height, :id]
    = f.hidden_field attribute, :id => attribute
  - if @object.is_a?(Profile)
    = hidden_field_tag :auto_crop_path,
      auto_crop_image_path(@object, @id)
  %br
  = f.submit "Rajaa alue", :class => "btn btn-primary"
```

Mitä tapahtuu, kun tämä lomake lähetetään? Tällä kertaa lomakkeen kohdepolkua ei muodosteta polkumetodilla, vaan määrittämällä toiminto eksplisiittisesti. Entä ohjain? Rails valitsee sen automaattisesti olion tyypin perusteella. Tässä tapauksessa olio on `@object`, jolle on annettu arvo `update_image`-toiminnossa:

```
@object = @profile
```

@object on siis Profile-olio. Niin ollen käytettävä ohjain on ProfilesController ja toiminto on profiles#crop_image.

Kuten näkyy, on varsin raskasta seurata, mitä edellä oleva skripti oikeastaan tekee. En käynyt esimerkkiä edes loppuun asti. Skriptin toimintaa seurattaessa on koko ajan siirryttävä näkymien, ohjaimien ja reittien välillä.

Ohjelmasuomessa oli maaliskuun aikana bugi, joka johtui siitä, että crop_image-toiminnolle välitetty ID oli väärä. Bugin etsintä vei minulta monta tuntia.

Toisaalta uskon, ettei koodin heikko seurattavuus ole ensisijaisesti Railsin vaan runsaan Ajaxin käytön vika. Ajax tekee koodista väistämättä vaikeampaa ymmärtää, ja täällä se on yhdistetty Railsin implisiittisiin lisäyksiin, kuten huomaamattomaan JavaScript-ajuriin. Lopputulosta on erittäin vaikeaa lukea.

Hyvä esimerkki tekemästäni isohkosta muutoksesta on automaattinen rajaus asiakasyritysten promokuville. Ellei asiakas rajaa kuvaa itse, Ohjelmasuomi rajaa sen, jotta kuva täyttää sille annetun tilan.

Toteutus alkaa Profile-mallin auto_crop_image-metodista:

```
def auto_crop_image(id)
  image = self.send(id)

  if id == "logo_header"
    size = '680x70#'
  else
    size = '200x200#'
  end

  image.thumb!(size)
  image.apply
  save
end
```

Send on metodi, joka kutsuu metodia, jonka nimi vastaa annettua symbolia tai merkkijonoa. Tällä metodilla sallitut muuttujan id arvot ovat ”logo_promo_1”, ”logo_promo_2”, ”logo_promo_3”, ”logo_promo_4” ja ”logo_header”. Auto_crop_image kutsuu vastaavan nimistä getteriä ja panee paluuarvon muuttujaan nimeltään image.

Image-muuttujan luokka on Dragonfly::ActiveModelExtensions::Attachment, joka on peräisin Dragonfly-gemistä (20). Metodi kutsuu olion thumb!-metodia, joka muuttaa kuvan kokoa. Risuaita size-merkkijonon lopussa tarkoittaa, että jos kuvasuhde ei vastaa haluttua, kuvaa on rajattava.

Lopuksi auto_crop_image tallentaa Profile-olion (mikä on tehtävä aina kuvan muuttamisen jälkeen).

Selvimmin tätä metodia yksinkertaistanut Rubyn ominaisuus on send-metodi. Monilla muilla ohjelmointikielillä reflektiota ei ole ollenkaan tai sitä on paljon vaikeampaa käyttää, jolloin yksinkertaisin toteutus edellyttää switch-lohkoa tai jopa useita if-lohkoja.

Toinen hyödyllinen ominaisuus on Dragonfly-gemin thumb!-metodi, joka sekä rajaa kuvaa että muuttaa sen kokoa. Tosin vastaavia kirjastoja voi hyvinkin olla saataville muille ohjelmistokehyksille.

Seuraava toteutuksen osa on toiminto:

```
def auto_crop_image
  if current_user.has_role?(:admin)
    profile = Profile.find(params[:profile])
  else
    profile = current_user.organization.profile
  end

  profile.auto_crop_image(params[:id])
  render :nothing => true
end
```

Koska ylläpitäjä voi muokata kenen tahansa profiilia mutta asiakas vain omaansa, toiminto tarkistaa, onko käyttäjä ylläpitäjä. Jos kyllä, toiminto hakee profiilin tietokannasta HTTP-pyynnössä olleen parametrin perusteella. Jos ei, toiminto hakee profiilin tietokannasta sen perusteella, kuka pyynnön lähetti.

Sen jälkeen toiminto kutsuu profiilin auto_crop_image-metodia. Lopuksi toiminto määrittää, ettei mitään piirretä, jolloin palvelin lähettää HTTP-headerit, muttei mitään muuta. Viimeksi mainittu määritys on tarpeen, koska Rails piirtää oletuksena toiminnon nimeä vastaavan näkymän.

Toiminto on suunnilleen yhtä yksinkertainen kuin se olisi esimerkiksi Spring-ohjelmistokehyksellä toteutettuna. Toiminnossa Railsin paras puoli on helppo pääsy HTTP-pyynnön parametreihin.

Seuraavaksi reitti:

```
match 'profiles/:profile/auto_crop_image/:id' =>
  'profiles#auto_crop_image', :as => 'auto_crop_image'
```

Reitin määritys on hyvin lyhyt ja selkeä.

Sitten tulikin kohdalle ongelma: toimintoa on kutsuttava Ajaxilla, kun asiakas sulkee kuvan lähetysikkunan. Niin ollen CoffeeScriptin on tiedettävä profiilin ID ja kuvan nimi.

Tietoja voi välittää skripteille esimerkiksi näkymättömillä elementeillä ja HTTP-headereilla. Valitsin piilotetun lomakekentän. Samoin päätin, että välitän skriptille suoraan reitin polun, jolloin reittiä voi muuttaa myöhemmin muuttamatta skriptiä.

Kävin läpi, mitä tapahtuu, kun asiakas lähettää promokuvan (kuvattu aiemmin tässä luvussa). Sinä aikana piirretään kaksi partiaalia: `app/views/profiles/_imagemodal.html.haml` ja `app/views/shared/_crop_thumbnails.html.haml`. Päätin panna reitin polun `crop_thumbnails`-partiaaliin:

```
- if @object.is_a?(Profile)
  = hidden_field_tag :auto_crop_path,
    auto_crop_image_path(@object, @id)
```

Tämä osa oli hyvin vaivalloinen toteuttaa, koska jouduin etsimään mallineet, jotka piirretään promokuvan lähettämisen aikana.

Viimeiseksi kirjoitin ne koodirivit, jotka kutsuvat toimintoa Ajaxilla, kun asiakas sulkee kuvan lähetysikkunan. Luin `imagemodal`-partiaalin lähdekoodin ja huomasin, että kummallakin elementillä, joka sulkee ikkunan, on HTML-määre `data-dismiss="modal"`.

```
$("[data-dismiss=modal]").click ->
  $.ajax({
    url: $("#auto_crop_path").val()
    type: "POST"
  })
```

`$("[data-dismiss=modal]")` on jQuery-valitsin. Se palauttaa jQuery-elementin, joka on sidottu niihin DOM-elementteihin, joilla on HTML-määre `data-dismiss="modal"`.

Skripti rekisteröi tapahtuman käsittelijän, jota kutsutaan, kun jompaakumpaa elementtiä painetaan. Tapahtuman käsittelijä yksinkertaisesti lähettää Ajax-pyynnön, joka kutsuu toimintoa. Pyynnön URL selvitetään dynaamisesti:

```
url: $("#auto_crop_path").val()
```

`$("#auto_crop_path")` palauttaa jQuery-elementin, joka on sidottu DOM-elementtiin, jonka ID on `auto_crop_path`. Kyseinen elementti on piilotettu lomakekenttä, jonka edellisellä sivulla oleva koodiesimerkki on luonut. `Val()`-funktio yksinkertaisesti palauttaa kentän arvon eli tässä tapauksessa reitin polun.

JQuery yksinkertaistaa tätä koodiesimerkkiä joka puolella. Ilman sitä koodi olisi huomattavasti pidempää ja monisanaisempaa ja sisältäisi pitkiä funktionimiä, kuten `querySelectorAll()` ja `addEventListener()`.

8 Yhteenveto

Tässä opinnäytetyössä pohdin, kuinka helppoa on jatkokehittää olemassa olevaa Ruby on Rails -ohjelmistokehyksellä toteutettua verkkopalvelua. Tutkin paitsi Rails-ohjelmistokehyksen, myös Ruby-ohjelmointikielen ja Railsin kanssa yleisesti käytettävien kirjastojen, kielten ja teknologioiden vaikutusta jatkokehitettävyyteen.

Yhteenvetona edellisestä luvusta Ruby on Rails -verkkopalvelu on jatkokehitettävyydeltään epätasainen kokonaisuus. Ainakin jos Ajaxia käytetään runsaasti, on vaikeaa seurata, mitä CoffeeScript-skriptit oikeastaan tekevät. Kehittäjän on jatkuvasti hypittävä näkymien, reittien ja ohjainten välillä. Lisäksi ainakin Ohjelmasuomen koodissa hyödynnetään Railsin implisiittisiä lisäyksiä, kuten huomaamatonta JavaScript-ajuria, mikä vaikeuttaa koodin seuraamista pahasti.

Parhaimmillaan Rails on palvelinpuolen koodissa. Esimerkkinä käyttämässäni metodissa `Profile#auto_crop_image` Rubyn `send`-metodi mahdollisti `switch`- ja `if`-lohkojen välttämisen. Samoin esimerkkinä käyttämäni reitti on hyvin lyhyt ja selkeä.

Epätasainen jatkokehitettävyys ei ole hyväksi, koska hyppiminen näkymien, reittien ja ohjainten välillä vie paljon enemmän aikaa kuin Railsin vahvuudet säästävät. Niin ollen en suosittele Railsia pitkiin projekteihin, joiden kehittäjien uskotaan vaihtuvan.

Lähteet

1 Ruby on Rails. 2013. Verkkosivusto. <http://rubyonrails.org/>. Luettu 26.4.2013.

2 Mornini, Tom. 2011. Here's Why Ruby On Rails Is Hot. Verkkodokumentti. Business Insider. <http://articles.businessinsider.com/2011-05-11/tech/30035869_1_ruby-rails-custom-software>. 11.5.2011. Luettu 1.2.2013.

3 King, Ryan. 2009. Kehittäjä, Twitter, San Francisco, Yhdysvallat. Blogikommentti 24.9.2009. "We use Scala for a few things at Twitter, but the majority of the site is Ruby." <http://blog.evanweaver.com/2009/09/24/ree/#comment-8291>. Luettu 1.2-.2013.

4 Viestintätoimisto CRE8 Oy. Verkkosivusto. <http://cre8.fi/>. Luettu 26.4.2013.

5 Ohjelmasuomi. Verkkosivusto. <http://www.ohjelmasuomi.fi/>. Luettu 26.4.2013.

6 Haikala, Ilkka & Märijärvi, Jukka. 2006. Ohjelmistotuotanto. Helsinki: Talentum.

7 ISO/IEC 25010. 2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE). Geneve: International Organization for Standardization.

8 Ruby, Sam & Thomas, Dave & Hansson, David Heinemeier. 2011. Agile Web Development with Rails: Fourth Edition. Pragmatic Programmers, LLC.

9 Sosinski, Robert. 2009. The Difference Between Ruby Symbols and Strings. Verkkodokumentti. <http://www.robertsosinski.com/2009/01/11/the-difference-between-ruby-symbols-and-strings/>. 11.1.2009. Luettu 19.2.2013.

10 Ruby on Rails. 2013. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/w/index.php?title=Ruby_on_Rails&oldid=534338370>. Luettu 23.1.2013.

11 Haml. 2013. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/w/index.php?title=Haml&oldid=541198115>. Luettu 22.3.2013.

12 Haml (HTML Abstraction Markup Language). 2013. Verkkodokumentti. <http://haml.info/docs/yardoc/file.REFERENCE.html>. Luettu 3.4.2013.

13 Ruby on Rails Documentation. Verkkosivusto. <http://api.rubyonrails.org/>. Luettu 26.4.2013.

14 Ruby on Rails Guides: Action Controller Overview. 2013. Verkkodokumentti. <http://guides.rubyonrails.org/action_controller_overview.html>. Luettu 6.4.2013.

15 Ruby on Rails Guides: Rails Routing from the Outside In. 2013. Verkkodokumentti. <http://guides.rubyonrails.org/routing.html>. Luettu 6.4.2013.

16 RSpec. Verkkosivusto. <http://rspec.info/>. Luettu 26.4.2013.

17 Buckley, Kerry. 2011. RSpec. Verkkodokumentti. <http://kerryb.github.com/iprug-rspec-presentation/>. 4.1.2011. Luettu 9.3.2013.

18 Sass (stylesheet language). 2013. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/w/index.php?title=Sass_(stylesheet_language)&oldid=541001009>. Luettu 22.3.2013.

19 JQuery. 2013. Verkkodokumentti. Wikipedia. <http://en.wikipedia.org/w/index.php?title=JQuery&oldid=546544494>. Luettu 24.3.2013.

20 Dragonfly (v 0.9.14). Verkkosivusto. <http://markevans.github.io/dragonfly/>. Luettu 26.4.2013.